Machine Learning Engineering

by Stas Bekman



Machine Learning Engineering Open Book

This is an ebook version of Machine Learning Engineering Open Book by Stas Bekman generated on 2026-10-01.

As this book is constantly being updated, if you downloaded it as a pdf or an epub file and the date isn't recent, chances are that it's already outdated - make sure to check the latest version at https://github.com/stas00/ml-engineering.

Machine Learning Engineering Open Book

This is an open collection of methodologies, tools and step by step instructions to help with successful training and fine-tuning of large language models and multi-modal models and their inference.

This is a technical material suitable for LLM/VLM training engineers and operators. That is the content here contains lots of scripts and copy-n-paste commands to enable you to quickly address your needs.

This repo is an ongoing brain dump of my experiences training Large Language Models (LLM) (and VLMs); a lot of the know-how I acquired while training the open-source BLOOM-176B model in 2022 and IDEFICS-80B multi-modal model in 2023, and RAG models at Contextual.AI in 2024.

I've been compiling this information mostly for myself so that I could quickly find solutions I have already researched in the past and which have worked, but as usual I'm happy to share these notes with the wider ML community.

Table of Contents

Part 1. Insights

1. The AI Battlefield Engineering - what you need to know in order to succeed.
2. How to Choose a Cloud Provider - these questions will empower you to have a successful compute cloud experience.
3. When Is It Worth Upgrading GPUs? - a practical framework for deciding whether a GPU generation upgrade is worth its cost, worked through on a real H200 → B200 benchmark.

Part 2. Hardware

1. Compute - accelerators, CPUs, CPU memory.
2. Storage - local, distributed and shared file systems.
3. Network - intra- and inter-node networking.

Part 3. Orchestration

1. Orchestration Systems - managing containers and resources
2. SLURM - Simple Linux Utility for Resource Management

Part 4. Training

1. Training - model training-related guides

Part 5. Inference

1. Inference - model inference insights

Part 6. Development

1. Debugging and Troubleshooting - how to debug easy and difficult issues
2. And more debugging
3. Testing - numerous tips and tools to make test writing enjoyable

Part 7. Miscellaneous

1. Resources - LLM/VLM chronicles

Updates

I announce any significant updates on my twitter channel https://twitter.com/StasBekman.

Ebook versions of the book

You can download various ebook formats of this book:

- PDF
- EPUB

I will try to rebuild these once in a few weeks or so, but if you want the latest ebook versions, the instructions for building are here.

Thanks to HuggingFace for giving me permission to host my book's ebook formats at the HF hub.

Skills for AI agents

I maintain two SKILL.md files that you can use to teach your AI agent:

- Machine Learning Engineering — train and operate large-scale ML models.
- Evaluate a GPU cluster — trial-node / acceptance loop: compute, network, and storage, then a markdown report.

See also the companion skills: The Art of Debugging and Stas' Python Cookbook.

Courses

- Lessons Learned from Training LLMs - provides a very different way of reading my open books, by going over the terse learned insights and allowing you to quickly dive deeper when you need to.

Lectures/Talks

- Building resilient ML Engineering skills given on 2026-01-10 for the GPU Mode community. Only had time to discuss performance reality of accelerators, network and storage and how each of them can be crucial to the ensemble's performance. Thanks to Mark Saroufim for organizing and providing an awesome support during the talk.

Discussions

If you want to discuss something related to ML engineering this repo has the community discussions available - so please don't hesitate to share your experience or start a new discussion about something you're passionate about.

Key comparison tables

High end accelerators:

- Theoretical accelerator TFLOPS
- Accelerator memory size and speed

Networks:

- Theoretical inter-node speed
- Theoretical intra-node speed

Shortcuts

Things that you are likely to need to find quickly and often.

Tools:

- all_reduce_bench.py - a much easier way to benchmark network throughput than nccl-tests.
- torch-distributed-gpu-test.py - a tool to quickly test your inter-node connectivity
- mamf-finder.py - what is the actual TFLOPS measurement you can get from your accelerator.

Guides:

- debugging pytorch applications - quick copy-n-paste solutions to resolve hanging or breaking pytorch applications
- slurm for users - a slurm cheatsheet and tricks
- make tiny models/datasets/tokenizers
- LLM/VLM chronicles collection

Gratitude

None of this would have been possible without me being entrusted with doing the specific LLM/VLM trainings I have learned the initial know-how from. This is a privilege that only a few enjoy due to the prohibitively expensive cost of renting huge ML compute clusters. So hopefully the rest of the ML community will vicariously learn from these notes.

Special thanks go to Thom Wolf who proposed that I lead the BLOOM-176B training back when I didn't know anything about large scale training. This was the project that catapulted me into the intense learning process. And, of course, HuggingFace for giving me the opportunity to work full time on BLOOM-176B and later on IDEFICS-80B trainings.

Recently, I continued expanding my knowledge and experience while training models and building scalable training/inference systems at Contextual.AI and I'm grateful for that opportunity to Aman and Douwe.

I'd also like to thank the numerous contributors who have been making this text awesome and error-free.

Contributing

If you found a bug, typo or would like to propose an improvement please don't hesitate to open an Issue or contribute a PR.

Related Open Books

- The Art of Debugging Open Book — methodologies and recipes for debugging Unix, Python and PyTorch programs.
- Stas' Python Cookbook — everyday Python and standard-library recipes.

License

The content of this site is distributed under Attribution-ShareAlike 4.0 International.

Citation

```
@misc{bekman2024mlengineering,
  author = {Bekman, Stas},
  title = {Machine Learning Engineering Open Book},
  year = {2023-2026},
```

```
  publisher = {Stasosphere Online Inc.},
  journal = {GitHub repository},
  url = {https://github.com/stas00/ml-engineering}
}
```

My repositories map

✔ Books: Machine Learning Engineering | The Art of Debugging | Stas' Python Cookbook

✔ Applications: ipyexperiments

✔ Tools and Cheatsheets: bash | conda | git | jupyter-notebook | make | python | tensorboard | unix

✔ Other Machine Learning: ML ways | Porting

The AI Battlefield Engineering - What You Need To Know

This chapter is one person's opinionated overview of the ML/AI Engineering reality, which may or may not be another person's reality. The intention is to help you start asking the right questions and get your ML Engineering needs met.

Basics

What's important in the AI race?

Training:

1. How fast one can train a better model (first to market advantage)
2. How much $$ was spent (do we still have money left to pay salaries to talent after training?)

Inference:

1. Fast latency (users are used to msec response times and will leave if the response takes seconds)
2. Fast throughput (how many concurrent queries can be processed)
3. How much $$ is being spent per user (can we rent more GPUs to acquire more users and/or improve (1) and (2)?)

What are the needs of LLM training?

1. Fast compute massively dominated by matrix multiplications
2. Fast enough memory, IO, network and CPU to feed the compute

Corollary: If when you buy or rent hardware you invest in the fastest accelerators, but cheap out on any of the other components you wasted $$ and you might not win the race as it'll take longer to train.

What are the workhorses of ML?

- An accelerator or a processing unit is what does most of the work.
- Since ML does a lot of parallel processing (SIMD) GPUs were used at the beginning, but now you additionally have TPUs, FPGAs, HPUs, QPUs, RDUs, etc. Recent CPUs are becoming used as accelerators as well, especially for inference.

More details.

AI driving entities

- AI companies - train models/build products around self-trained or trained-by-others' models, in-house research.
- Academia - does massive research and write papers. Lots of new ideas are generated.
- AI enthusiasts - lots of good will available, some pull resources/talents together to train open access models, with donated compute by HPCs and an occasional cloud, or a university cluster.
- Entrepreneurs - lots of low hanging fruit to pick - creative reselling of services, making ML-driven apps, and using various ingenious combinations of available resources to create amazing outcomes.

Information sharing

- It's very surprising that almost everybody involved in the domain of AI shares a lot of the discoveries with

the community.
- Surely, companies don't disclose all of their IP, but a lot of it does get shared in the form of knowledge or model weights
- Companies that publish a lot of IP and models tend to attract higher quality talent.
- Twitter seems to be the central platform where one must be to follow what's going on

The AI bubble

- The Dot-com bubble occurred during 1995-2000. And a very similar situation is happening right now in the AI space.
- There is a lot of money available to create new startups or boost the existing companies. It's relatively easy to raise millions of dollars.
- As we are in the wild-wild-west stage of the AI industry it's very difficult to predict the future, and so pretty much anything goes as far as startup ideas go, as long as it sounds reasonable.
- What distinguishes the AI bubble from the Dot-com bubble, is that one didn't actually need much money to operate a Dot-com company - most of the raised money went to marketing and some to staff, barely any to compute. AI companies need millions of dollars because training LLMs requires an insane amount of compute, and that compute is very expensive. e.g. 1x NVIDIA H100 costs ~$30k and a company may need 512 of those, which is $15M (not counting the other hardware components and related costs)!

ML Engineer's heaven and hell

This is my personal LLM/VLM trainings-based heaven and hell. YMMV.

ML Engineer's heaven

1. A well built HPC, or a full service cloud based cluster, where someone diligently and timely takes care of the hardware and the systems.

   I just need to bring my training software and do the training, which is already an insanely complicated job requiring special skills.
2. Lots of nodes available for exclusive unlimited use
3. Fast inter-node connectivity that doesn't bottleneck the accelerators and which isn't shared with other users
4. Huge local super-fast NVME based shared filesystem that can fit datasets and checkpoints
5. Barebones Linux w/ SLURM and minimal software to be able to launch training jobs
6. sudoer access to ease the work with a team of people

ML Engineer's hell

1. A cloud or in-house cluster, where you have to do everything - sysadmining, replacing hardware, dealing with outages, etc. And to do the training on top of that.
2. A smallish slow shared filesystem (NFS?), with cloud to draw data from and checkpoint to
3. Slow inter-node leading to low accelerator utilization
4. Inter-node shared with other users which make the network erratic and unpredictable
5. Super-complicated cloud console with gazillion of screens and steps to set even simple things up
6. Not being able to swap out failing hardware fast

7. Needing to timeshare the nodes - with wait times between training jobs
8. Having other concurrent users who might use up the whole disk, leading to trainings crashing
9. Not being able to kill jobs others on the team started and went to sleep

Getting compute

There are 3 main choices to where one gets compute:

- Rent on the cloud
- Get a timeshare on an HPC
- Buy it

Renting on the cloud

This is currently the prevalent way of getting compute.

Pros:

- Easy to expand or contract the size of the cluster
- Easy to upgrade from the old hardware generation to the new one in a few years
- Cluster management could be easily outsourced

Cons:

- Expensive, unless you negotiate a long term (1-3 year) contract for hundreds of accelerators
- You will be tempted to buy many tools and services that you may or may not need
- You always get charged whether you use your cluster fully or not

Using HPC

There aren't that many HPCs out there and so the amount of available resources is limited.

Pros:

- Managed for you - all you need is your software to do the training and a bit of SLURM know-how to launch jobs
- Often sponsored by the local government/university - probably could get the job done for less $$ or even free (e.g. we trained BLOOM-176B for free on JeanZay HPC!)

Cons:

- needing to time share compute with other teams == short job times with possible long wait times in between - could be difficult to finish training quickly
- The inter-node network is likely to be unstable as it'll be used by other teams
- Have to abide by the HPC's rules (e.g. no `sudo` access and various other rules to follow)
- In a way the HPC cluster will be what it'll be - you can't make the network faster and often even getting some software installed can be tricky.

Buying hardware

It's mainly universities that buy and build their own clusters, and some big companies do that too.

Pros:

- If you can deploy the hardware 24/7 for more than a few years the total cost will be cheaper than renting
- Easy to provide fast local storage - a good NVME raid would be much cheaper and faster than online storage

Cons:

- You're stuck with the outdated hardware just a few years after it was purchased - might be able to resell
- Must buy more than needed - Hardware tends to break, especially when it's used 24/7, RMA could take weeks
- Have to hire talent to manage the in-house solution
- Have to figure out cooling, electric costs, insurance, etc.

Managing compute

- Unless you use a fully managed HPC compute you absolutely need to hire a sysadmin. It may feel that your ML engineers can swing that between their training jobs, but they will be losing a lot of time to managing disk space, dealing with problematic nodes, asking users to behave, etc.

The needs of technology

Can you feed the furnace fast enough?

Imagine a steam locomotive - the engine is great, but if the fireman isn't fast enough to shovel the coal in, the train won't move fast.



source

This is the current state of ML hardware: The bottleneck is in moving bits and not the compute.

- Accelerators get ~2x faster every 2 years (Moore's law)
- Network and memory are not! Already now both are compute bottlenecks

- IO can be another bottleneck if your DataLoader has to pull data from the cloud
- CPU is fine as long as it has enough cpu-cores for DataLoader workers, and main processes

Corollary: research the whole machine and not just its engine.

a crazy idea: the older GPUs might do fine if you can actually feed them as fast as they can compute. And if you can get 3x of them at the same cost as the next generation GPU you might finish training sooner and a lower cost.

TFLOPS

- Once you choose the architecture and the size of the model and how many tokens you want to train the model for you immediately know how much compute will be required to accomplish this goal. Specifically you can now calculate how many floating point operations will be needed.
- All that is missing is comparing different compute providers to how many floating point operations their hardware can computes per secs (TFLOPS) and their cost per unit and now you can tell the total approximate cost of the training.

    1. Calculate the time needed to train given the TFLOPS of the considered solution: `total_tflops_required / tflops_of_this_compute_unit = time_in_seconds` Let's say it came to be 604800 secs or 7 days.
    2. Look at the cost of using this compute solution for 7 days and now you know the total $$ to train this model.
    3. Look at other proposals and calculate the same - chose the best option.

- As mentioned earlier, time is of a huge importance, so you might still choose a more expensive solution if finishing the training sooner is important because you want to be first to market.

Unfortunately, this math is only partially correct because the advertised peak TFLOPS are typically unachievable. The MFU section delves into it.

Model FLOPS Utilization (MFU)

As mentioned in the previous section, some (most?) vendors publish unrealistic peak performance TFLOPS - they aren't possible to achieve.

Model FLOPS Utilization (MFU) is the metric that tells us how well the accelerator is utilized. Here is how it is calculated:

1. Measure the actual TFLOPS by calculating how many floating point operations a single training iteration takes and dividing that number by the number of seconds this iteration took.
2. Divide the actual TFLOPS by advertised TFLOPS to get the MFU

Example: Let's say you're training in BFLOAT16 precision:

- If a single iteration requires 624 Tera floating point operations and it took 4 secs to run then we know that we get: `624/4=156` actual TFLOPS
- now BF16@A100 is advertised as 312TFLOPS so `156/312=0.5` gives us 50% MFU.

Practically:

- with NVIDIA GPUs if you're above 50% MFU on a multi-node setup with a large model you're already doing fantastic
- recent advancements in more efficient scalability solutions keep on increasing MFU
- slow networks and inefficient frameworks or untuned configuration lower MFU

Therefore once you know the MFU you can now adjust the cost estimate from the previous section. In the example there we said it'll take 7 days to train, but if MFU is 50%, it means it'll take 14 days to train.

Moving bits

Why can't the advertised TFLOPS be achieved? It's because it takes time to move data between accelerator memory and compute and additionally it takes even more time to move data from disk and other gpus to the accelerator's memory.

- There is not much that can be done about the accelerator memory since its bandwidth is what it is - one can only write more efficient software to make data move faster to/from the accelerator - hint: fused and custom written kernels (like torch.compile and flash attention)
- If you only have a single GPU and the model fits its memory, you don't need to worry about the network - accelerator memory is the only bottleneck. But if you have to shard the model across multiple GPUs network becomes the bottleneck.
- Intra-node Network - is very fast, but difficult to take advantage of for large models - Tensor parallelism and sequence parallelism address part of this problem. (more).
- Inter-node Network - typically is too slow on most server setups - thus this is the key component to research! Efficient frameworks succeed to partially hide the comms overhead by overlapping compute and comms. But if comms take longer than compute, the comms are still the bottleneck. more.
- Storage IO is important primarily for feeding the DataLoader workers and saving the checkpoints. more.
    1. Typically with enough DL workers the DataLoader adds very little overhead.
    2. While checkpoints are being saved the accelerators idle unless some async saving solution is used, so fast IO is crucial here

Key hardware components

Accelerators

As of 2026-08 here are the most common accelerators that can be used for training, finetuning and inferencing ML models:

Widely available:

- NVIDIA H200, B200, B300 - B300 systems are shipping
- AMD MI325X, MI350X and MI355X - neoclouds primarily, but large CSPs have started offering AMD too
- Intel Gaudi3 - on Intel's cloud and via OEM servers

Available, but locks you in:

- Google TPUs - fast, but a lock-in to one vendor and one cloud, and the software path is XLA rather than CUDA. It appears Google started to sell TPUs to some companies (outside of their cloud) - not officially confirmed, but it would weaken the lock-in argument if it holds
- AWS Trainium2 - on AWS, also XLA
- Cerebras WaferScale Engine - on Cerebras' cloud

Not yet available:

- NVIDIA Rubin - specs published but still marked preliminary
- AMD MI455X - Helios volume deployments expected 2H-2026
- SambaNova SN50 - shipping "in the second half of 2026"; SN40L is the current part
- Intel Crescent Island - inference GPU, customer sampling announced for H2-2026; no published TFLOPS

Regional:

- Huawei Ascend - real scale, but China-only in practice, and the specs are only on Huawei's Chinese pages

Beware that "announced" is not "available" - the Accelerators tables are split in two for exactly this reason, `Generally available` first and `Announced, availability not confirmed` after it. For the full list and the numbers see Accelerators.

Accelerator Interoperability

In general most (all?) accelerators are supported by major frameworks like PyTorch or TensorFlow and the same code should run everywhere with small modifications as long as it doesn't use any accelerator-specific functionality.

For example, if your PyTorch application calls `torch.mm` - it should work everywhere, but if it includes custom CUDA kernels it'll only work on NVIDIA GPUs and may be on the recent AMD MI-series.

- NVIDIA GPUs: all based on CUDA, which most training frameworks support. You can easily moved between different NVIDIA GPUs and most things would work the same.
- AMD MI250/MI3xx: with PyTorch using ROCm you can run most CUDA-based software as is. This is really the only inter-operable accelerator with the NVIDIA stack.
- Intel Gaudi2/Gaudi3: if you use HF Transformers/Diffusers you can use optimum-habana. If you use HF Trainer with NVIDIA GPUs it should be relatively easy to switch to train/infer on Gaudi2.
- Google TPU and AWS Trainium: both via XLA - PyTorch works, custom CUDA kernels don't
- Huawei Ascend: via CANN, with MindSpore first-party and PyTorch adaptation alongside it
- Cerebras: is also working on PyTorch support via Cerebras Software Platform (CSoft) via XLA.

Also in general most ML code could be compiled into cross-platform formats like Open Neural Network Exchange (ONNX) which can be run on a variety of accelerators. This approach is typically used more often for inference workloads.

Network

- If you want to train a large model that doesn't fit onto a single accelerator's memory you have to rely on the intra- and inter-node networks to synchronize multiple accelerators.
- The biggest issue right now is that compute hardware advancements move faster than networking hardware, e.g. for NVIDIA NVLink intra-node (unidirectional bandwidth):

GPU	Compute fp16 TFLOPS	Compute speedup	Intra-node GBps	Intra-node speedup
V100	125	1	150	1
A100	312	2.5	300	2
H100	989	8	450	3
B200	2250	18	900	6
Rubin	4000	32	1800	12

- You can see that A100 was 2.5 faster than V100, and H100 is ~3x faster than A100. But the intra-node speed of NVLink has only increased by 150GBps each generation. NVLink 5.0 doubled the speed over NVLink 4.0 so it catches up a little bit with the compute speed ups. But the speed up is still insufficient.
- Moreover, each NVLink generation raised both the per-link bandwidth (2.5 → 3.125 → 6.25 → 12.5 GBps for NVLink 1–4, then 25 GBps for NVLink 5) and, in most generations, the number of links, so per-GPU bandwidth climbed from 80 GBps (P100) to 450 GBps (H100) to 900 GBps (B200).
- The inter-node situation isn't any better with most NICs there doing 100 or 200Gbps, and some 400Gbps are starting to emerge. (correspondingly in GBps: 12.5, 25 and 50). It's the same story here, some solutions

provide dozens of NICs to get to higher speeds.

- Also typically with LLMs the payload is so large that network latency is often negligible for training. It's still quite important for inference.

Intra-node Network

- Pay attention to bytes vs bits. 1Byte = 8bits. 1GBps = 8Gbps.
- If you need to reduce bits (e.g. gradients) across multiple nodes, it's the slowest link (Inter-node) that defines the overall throughput, though hierarchical algorithms sometimes can speed things up taking advantage of the faster intra-node bandwidth
- Tensor parallelism and sequence parallelism have to remain within the node to be efficient - only makes sense with fast intra-node speed

NVIDIA:

- Some have a lot of NVLinks, others less, but typically plenty with 1800GBps (14.4Tbps) unidirectional bandwidth for Rubin, 900GBps (7.2Tbps) for B200/B300, 450GBps (3.6Tbps) for H100/H200, 300GBps for A100 nodes.

Intel Gaudi2:

- 300Gbps card-to-card (100GbE RoCE v2 RDMA; same NICs also serve inter-node)

More details

Inter-node Network

- You will see a wide range of speeds from 200Gbps to 6400Gbps
- Inter-node links remain about an order of magnitude slower than intra-node ones (e.g. 50GBps per accelerator of EFA v4 on a P6-B200 instance vs 900GBps of NVLink 5), but the practical gap is far smaller than that, because an inter-node collective also does most of its data movement over the intra-node links - a 4GiB `all-reduce` across 4 nodes measures only ~2x slower than on one node, not 18x. See Inter-node speed depends on intra-node speed
- You need to reduce gradients and other tensors (e.g. MoE) faster than compute to avoid idling accelerators
- You typically get at most 80% of advertised speed. e.g., if you are told you get 800Gbps, expect ~640Gbps

More details.

Storage

There are 3 distinct Storage IO needs in the ML workload:

1. You need to be able to feed the DataLoader fast - (super fast read, don't care about fast write) - requires sustainable load for hours and days
2. You need to be able to write checkpoints fast - (super fast write, fastish read as you will be resuming a few times) - requires burst writing - you want super fast to not block the training for long (unless you use some sort of cpu offloading to quickly unblock the training)
3. You need to be able to load and maintain your codebase - (medium speed for both reading and writing) - this also needs to be shared since you want all nodes to see the same codebase - as it happens only during the start or resume it'll happen infrequently

- Most of the time you're being sold 80% of what you paid. If you want a reliable 100TBs you need to rent 125TBs or your application may fail to write long before the disk is full.

- Shared Distributed Filesystem:
    1. non-parallel shared file systems can be extremely slow if you have a lot of small files (=Python!)
    2. You want Parallel FS like GPFS (IBM Spectrum Scale) or Lustre (Open Source)

More details.

CPU Memory

You need enough memory for:

- 2-3 possibly DL workers per Accelerator (so 16-24 processes with 8 accelerators per node)
- Even more memory for DL workers if you pull data from the cloud
- Enough memory to load the model if you can't load to accelerator directly
- Often used for accelerator memory offloading - extends accelerator's memory by swapping out the currently unused layers - if that's the target use, then the more cpu memory is available - the better!

CPU

This is probably the least worrisome component.

- Most clouds provide beefy CPUs with plenty of cpu cores
- You need to have enough cores to run 2-3 DL workers +1 per gpu - so at least 30 cores
- Even more cores for DL workers if you have complex and/or slow DL transforms (CV)
- Most of the compute happens on GPUs

Impress others with your ML instant math

Tell how many GPUs do you need in 5 secs

- Training in half mixed-precision: `model_size_in_B * 18 * 1.25 / gpu_size_in_GB`
- Inference in half precision: `model_size_in_B * 2 * 1.25 / gpu_size_in_GB`

That's the minimum, more to have a bigger batch size and longer sequence length.

Here is the breakdown:

- Training: 8 bytes for AdamW states, 4 bytes for grads, 4+2 bytes for weights
- Inference: 2 bytes for weights (1 byte if you use quantization)
- 1.25 is 25% for activations (very very approximate)

For example: Let's take an 80B param model and 80GiB GPUs and calculate how many of them we will need for:

- Training: at least 23 GPUs `80*18*1.25/80`
- Inference: at least 3 GPUs `80*2*1.25/80`

More details.

Traps to be aware of

As you navigate this very complex AI industry here are some thing to be aware of:

Say no to "will make a reasonable effort to ..." contracts

- If you contract doesn't have clear deliverables (time and performance) don't be surprised if you paid for something you won't receive in time you need it or not at all
- Be very careful before you sign a contract that includes clauses that start with "we will make a reasonable effort to ...".

  When was the last time you went to the bread section of the supermarket and found a lump of half-baked dough with a note "we made a reasonable effort to bake this bread, but alas, what you see is what you get"?

  But for whatever reason it's acceptable to create a legal contract where the provider provides neither delivery dates nor performance metrics and doesn't provide stipulations for what will they do in recompense when those promises aren't fulfilled.

Beware of hardware and software lock-in scenarios

- Some cloud providers will make you use very proprietary tools or hardware that will make it very difficult for you to leave down the road because you will have to retool everything if you leave
- Consider what would be the cost of moving to a different provider should this provider prove to be not satisfactory or if they don't have a capacity to fulfill your growing needs.
- If you rent a cluster with a generic Linux box with generic open source tools it should be trivial to move from one provider to another as almost everything would work out of the box
- Obviously if you choose compute that requires custom software that works for that hardware only and you can't rent this hardware anywhere else you're setting yourself up for a lock-in

Don't buy what you don't really need

- The cloud providers have mostly the same generic hardware, which leads to a very slim $$ margin and so in order to make big $$ they invent products and then try to convince you that you need to buy them. Sometimes you actually need those products, but very often not. See also the previous section on lock-in, since proprietary products usually mean a partial lock-in.
- Often it's easy to observe the 3 step marketing technique for solutions that seek a problem to solve:

    1. Convince a couple of well respected customers to use the provider's proprietary products by giving them huge discounts or even pay them to use them
    2. Use those in step 1 as the social approval lever to reel in more converts
    3. Then scoop the rest of the strugglers by telling them that 80% of your customers (1+2) use these amazing products

When marketing these products it's important:

- to mention how well they work with a dozen of other products, since now you're not buying into a single product but into a whole proprietary product-sphere.
- to use really nice looking complicated diagrams of how things plug into each other, and move really fast to the next slide before someone asks a difficult question.

HPCs are probably a good group of compute providers to learn from - they have no funds to create new products and so they creatively address all their needs using mostly generic open source tools with some custom written software added when absolutely needed.

Unsolicited advice

To conclude I thought I'd share some insights to how one could slightly improve their daily AI battlefield experience.

FOMO and avoiding depression

If you read Twitter and other similar ML-related feeds you're guaranteed to feel the fear of missing out, since there is probably at least one new great model getting released weekly and multiple papers are getting published daily and your peers will publish their cool achievements every few minutes.

We are dealing with **very complex** technology and there is a small handful of people who can absorb that much new material and understand / integrate it.

This can be extremely depressing and discouraging.

I deal with it by looking at twitter about once or twice a week. I mostly use Twitter in broadcast mode - that is if I have something to share I post it and only watch for possible follow up questions.

Usually all the important news reach me through other people.

Don't try to know everything

The pace of innovation in the field of AI is insane. It's not possible to know all-things-AI. I'd dare to say it's not possible to know even 10% of it for most of us.

I realized this very early one and I stopped paying attention to most announcements, tutorials, keynotes, etc. Whenever I have a new need I research it and I discover what I need and I have to be careful not to try to learn other things not pertinent to the goal at hand.

So I actually know very little, but what I have researched in depth I know quite well for some time and later I forget even that (that's why I write these notes - so that I can easily find what I have already researched).

So if you ask me something, chances are that I don't know it, but the saving grace for me is that if you give me time I can figure it out and give the answer or develop a solution.

Don't beat yourself up when using half-baked software

Because the ML field is in a huge race, a lot of the open source software is half-baked, badly documented, badly tested, at times poorly supported. So if you think you can save time by re-using software written by others expect spending hours to weeks trying to figure out how to make it work. And then keeping it working when the updates break it.

The next problem is that most of this software depends on other software which often can be just as bad. It's not uncommon where I start fixing some integration problem, just to discover a problem in a dependent package, which in its turn has another problem from another package. This can be extremely frustrating and discouraging. One tries to save time by code reuse, but ends up spending a long time figuring out how to make it work. At least if I write my own software I have fun and it's a creative process, trying to make other people's software work is not.

So at the end of the day we are still better off re-using other people's software, except it comes at an emotional price and exhaustion.

So first of all, try to find a way not to beat yourself up if the software you didn't write doesn't work. If you think about it, those problems aren't of your creation.

Learning how to debug efficiently should also make this process much less painful.

How to Choose a Cloud Provider

Having used multiple compute clouds over long and short terms, and participating in many "discovery" calls, I've learned that it's absolutely crucial to approach the cloud choosing process with an utmost care and dedication. Especially for the long term contracts - you may end up in a 3-year lock-in where you pay millions of dollars and end up having a terrible experience and no way to get out of the contract.

To give you a perspective - a 64-node cluster may easily cost USD$20-50M over a 3 year period. This is often more than what startups pay for the salaries.

I can't stress this enough that choosing a bad 3-year contract may prevent your startup from succeeding.

In this article I'm not going to tell which clouds to avoid, but instead try to empower you to avoid having a bad experience and to have at least a decent one, that will give your company a chance to succeed.

These notes assume you already know what compute you want for your specific workloads. If you don't please skim through the Accelerator, Storage and Network chapters to know what's available out there. Most of the time you want the latest the clouds have to offer.

Glossary

- CSP: Cloud Service Provider
- SLA: Service-level_agreement
- SLO: Service Level Objective
- TCO: Total Cost of Ownership

The most important questions to ask

In my limited experience there are 3 questions you absolutely need to know the answer to:

1. **Your GPUs will fail - guaranteed.** The critical question is how quickly they will be replaced and what's the procedure - if you need to get a hold of a human for each replacement this will be painful and slow, the best CSPs give you an API for you to do it yourself. Also beware of Humpty Dumpty GPUs, that will get returned to you w/o actually being replaced because when the technician checks them they appear to work just fine - but they will fail again when you try to use them - here is how to detect if you get the same broken GPUs back.
2. **You will run into multiple problems, mainly with the hardware.** You need to know how quickly you can reach an engineer that can address the issue. Normally your reports will go to a PM, who will triage and try to get you to the right person, but very often this can take hours/days and sometimes requires escalation, especially if the PM isn't the most experienced person in the world.
3. **You need to know if you can break the contract - how quickly and at what cost.** Things may get really bad for your team, but if you can't walk away, that would guarantee burnout for one or more persons on your team and a potential negative impact on your ability to make progress and lead your team to success.

Contracts

If you're paying per hour, you don't need to worry about contracts. But this method isn't good long term because you will be paying many times more and you won't have a steady reliable accelerator foundation. A long term contract at times and with a good negotiator can lead to a 10x in total cost of ownership (TCO) savings (and time)!

Free trials

Most cloud service providers (CSPs) have trial programs where you can "kick the tires" for a few days/weeks on a few nodes for free.

Granted, it won't give you an indication of how well the bigger cluster would scale, but it should be sufficient to be able to run quite a few benchmarks and experiments.

It also will give you a good opportunity to check how the provider's customer support works (if any support is included in the free package that is).

Half-baked solutions

Since a new generation of accelerators happens roughly every 12-18 months and the customer wants those latest accelerators "yesterday" to have a business advantage over their competitors - this gives CSPs barely any time to integrate the new generation of the hardware, test it, adapt their software stack and burn those components in.

So if you want the latest generation as soon as it becomes available you're almost guaranteed to have a bad experience because, well, time is needed to get things right - we are talking about months of waiting. But customers rule - so the CSPs give them what they want, often not quite telling that what the customer gets is not quite ready.

I'm not sure if CSPs are to blame, because often they get the hardware delivery months after it was promised by the manufacturers and, of course, by now they can't keep their promises to the customers, so they just go ahead and deliver...

Then some CSPs develop their own hardware (e.g. network stack) in order to have better margins and then they fail to complete those custom solutions in time, the latest accelerators are there, but the whole system is limping. It's much safer when off-the-shelf components are offered, since those are most likely to be well-tested working components (except it's likely to cost more).

I think it's OK if the customer wants the hardware early, there should just be an honest disclosure as in: *"look we need some 3 more months to make things solid, if you want the nodes now you can have them but we can't guarantee anything."*

We-will-do-our-best clause

A lot of the long-term cloud contracts are likely to include a lot of "we will do our best" clauses.

Yet:

1. The customer is not allowed to "do their best" to pay, they are legally obliged to pay the amount they agreed to pay and on time.
2. The customer is not allowed to break a contract before its term runs its course.

In my experience "we will do our best" is demonstrated by Tier-1 clouds by sending 10+ people to the meetings with the customers. Some of them will be clueless and will be just sitting there making the company look resourceful: *"look, we are allocating 10+ people to the problem you're experiencing. You have nothing to worry about"*. Except, most of the time those people can't solve your problem.

What you need is just 2 cloud support people on the call - one product manager and one engineer directly responsible for solving the problem at hand. And in my experience this sort of meeting could take weeks to months to manifest or not at all. Usually one needs to have good connections to be able to escalate the issue to "top brass".

For every critical component of the package you're purchasing you need a quantifiable delivery metric. For example, if the network you were sold is supposed to run at X GBps at that many nodes doing all-reduce, and you measured it to be significantly lower, there should be a stipulation of what the CSP will do when this happens. Same goes for storage, accelerators and any other critical component that you plan to rely on.

Of course, it's up to you to negotiate the specific repercussions, but probably the best one is that you stop paying until the problem is fixed. That way there is an incentive for the problem to be fixed.

Alas, not paying helps, but not being able to use the compute is still a huge problem. And breaking the contract and migrating to another provider is a huge undertaking not to be taken lightly. But at least there is something you could do if you don't get what you need.

I must also say that it's almost never the problem of the engineers, very often they are amazing experienced people - most of the time it's the issue of management and resource allocation. So please be as gentle as possible with the people you interact with, while firmly demanding a resolution. I know it's a difficult one - more than once I was at the end of the rope, and I couldn't always keep it cool.

Service Level Agreement

As a continuation of a previous section, a Service Level Agreement (SLA) is an agreement between a service providers and a customer that define various guarantees and expectations with regards to service quality and availability, and various responsibilities.

The other term is Service Level Objective (SLO) where SLA is quantified. For example, an SLO may define a Monthly Uptime Percentage to 99.5%, if the uptime is less than 99.5% the provider credits the customer to a certain percentage of the $$ spent. For example, 10% if the uptime is 99-99.5%, 25% for 95-99%, etc. Here is an example GCP SLA.

The main category one should care for when renting ML clusters is failing accelerators and/whole nodes. If you paid for 64 nodes but were able to use only 60 you should be reimbursed/credited for those nodes you couldn't use. Your SLA should define the duration of downtime after which the provider starts paying you back and how much.

Same goes for network and storage, albeit those typically fail a lot less often than accelerators, but they do fail.

In general any critical part of the service should have an SLO and clearly defined repercussions if the SLOs aren't met.

Most Tier 1 companies should already include their standard SLAs in the contract. In theory the customer should be able to negotiate those to adapt to their needs, though it might not always be possible. Sometimes offering to pay more may allow for a better than standard SLO.

Discuss a contract breaking clause

Both sides should be empowered to experience a mutually beneficial business experience.

Therefore it's critical that you should be able to legally exit the contract should your business experience not be beneficial because the other side is failing to meet the agreed upon expectations.

This, of course, implies not to have a legal battle which can be very costly and Tier-1 clouds have a lot of money to hire the best lawyers, so it might be a losing battle.

It's up to you to negotiate under which circumstances the contract can be cleanly exited before its term runs out.

Must have paid support included

In one of the companies I worked at our cloud contract didn't include the paid support service and the only support we had was via a customer chat. The paid support was skipped to save costs, but boy did we end up losing days of compute because of that.

Do not try to save here - you will end up losing a lot of money, developer time and hair. Make sure you have a way to submit tickets with priority labels and a defined in the contract expectation to how quickly they will be dealt with.

When you try to use customer chat to solve an urgent problem, there is zero obligation for them to do anything, or at least to do it in a timely manner.

If you're dealing with PMs, you need to know how quickly you could talk directly to the end-point engineer, while removing the middle-man.

Support during off-hours

Do you get human support for emergencies on weekends/holidays/nights? e.g. On one of the HPCs I used the human support was only available Mon-Fri 9-5.

If this is not available, at the very least ensure that your team can perform cluster resuscitation themselves - and do a drill to ensure this is actually doable. This means you need to have an API to perform all those things without the provider's support.

Next generation accelerator migration

On average a new generation of accelerators comes out every 12-18 months, but a typical contract is for 3 years. Which means that for about half of that time you will end up using an inferior product.

Nobody wants to use a 2-5x slower accelerator when a much faster version is available, but most customers now are stuck with the old accelerators for the full 3 year duration.

You need to negotiate the ability to move to the new generation before the end of the term, which would obviously require some additional money paid for this to happen.

Ensure all accelerators are at the same region/locale

As new accelerators emerge it's very often the case that if you want them early they won't be available in the same region as your current accelerators are. Unless you drop the previous allocation completely and move to a new one at a different region you will have a nightmare of maintaining multiple storage copies, because for performance reasons you need the storage to be where the accelerators are. If you don't you will have issues with syncing multiple copies of the same storage and paying potentially huge ongoing egress/ingress costs. So plan for that and discuss that the CSP moves your older generation accelerator allocation to the same region where the new generation is. That's, of course, not always possible and you may have to wait till more accelerators become available. But it's an important clause to discuss.

Accelerators

This group of questions/issues is specific to accelerators.

Accelerators need to be burned in

When a new batch of components arrives the provider has to "burn them in" before handing them to customers. This is a process of running an extensive stress testing to detect any accelerators and other system components that are faulty.

If this is not done, the customer ends up discovering the "bad apples" the hard way, while running their workloads. This leads to lost compute and developer time. If the workload uses a few nodes, one failing accelerator isn't a big problem most of the time, but if the workload uses dozens or hundreds of nodes the cost is huge.

It shouldn't be the responsibility of the customer to discover bad accelerators. And while there is no guarantee that the accelerator will not fail after it has been stress tested - it should happen rarely.

Otherwise, a new batch of accelerators often has a 3-10% failure rate, which is huge and very costly to the customer!

So ask your provider how long did they burn in your accelerators/systems for, if at all.

I'm yet to find a golden reference point, but, for example, SemiAnalysis suggests that OEM provider performs a 3-4 weeks burn-in, and then the CSP conducts another 2-3 day long burn-in/acceptance test. So if that's the case you want to ensure that the systems were stress-tested for at least 2-3 days. And, BTW, nothing stops you from doing some massive training as a test if you're not worries about stop-n-go for failures, if it works you made some progress. Of course, you want to make sure that your training stresses out all parts of the system in a sustainable fashion.

Dealing with accelerator failures

In my experience, while other compute components do fail occasionally, 95% of the time it's the accelerators that fail.

Therefore you need to have a very clear and quick path to an accelerator replacement.

Ideally this process needs to be automated. So you need to ask if there an API to release a broken node and get a replacement. If you have to ask a human to do that, it usually doesn't work too well. The more automated things are, the more efficient the experience.

How many accelerators do you have in the provider-side back up pool available to you? They will usually commit to a certain number of fast replacement per month.

That said, if time is of the essence to your workflows, you should always pay for about 10% more nodes than you need, since most of the time you won't be able to get instant replacements. The extra nodes can be used for development and if you have failing nodes during training you can instantly use your own extra nodes.

Ensure all your nodes are on the same network spine

Unless you're renting 10k gpus, most smaller clusters can easily be co-located on the same network spine - so that it takes the same time to perform inter-node network traffic from any node to any other node.

Ensure that any back up nodes that you're not paying for, but are there to deal with failing accelerators, reside on the same network spine as the nodes you're paying for. If they don't, you are going to have a big problem if you do multi-node training - since that one replacement node will be further away from all other nodes and will slow the ensemble down (the weakest link in the chain).

Ensure you keep your good accelerators on reboot

You want your cluster to have a fixed allocation. Which means that if you need to re-deploy nodes, and especially if you're planning a downtime, other customers aren't going to grab those nodes!

Once you spent weeks filtering out the bad nodes from the good nodes, it's crucial to keep those nodes to yourself and not start the painful and costly filtering again.

Do you think you will need to expand?

This is a difficult one, because it's hard to know ahead of time if the amount of nodes you're asking for will need to grow in the future.

Ideally you'd want to discuss this with your provider in case they could plan for your imminent expansion.

Because otherwise, say, you want to double the number of your nodes, but in order to get more nodes, they could only be allocated on another network spine - this is going to be a problem, as it'd impact the training speed.

Chances are that you will have to drop your current allocation and move to another bigger allocation - possibly even in a different region if they don't have local capacity. And moving to a different region can be a very slow and costly experience because you have to move your storage to where your new cluster is. Based on a personal experience - don't treat this lightly.

Storage

Large and fast storage is very important for both - good developer experience and fast training/finetuning/inference workloads - in particular with regards to loading/saving checkpoints.

Guaranteed maximum capacity

Ask how much of the storage you will be paying for is guaranteed.

For example, if the Lustre filesystem is used the customer needs to know that they have to over-provision by 25% to get the actual storage capacity they need, because Lustre can fail to write at 80% total storage capacity, because of bad disk balancing design. And the onus of paying for the extra 25% is on the customer!

Most other filesystems I had an experience with typically reach 100% capacity without failing, but it's always good to ask for the specific filesystem you plan to use.

Know your storage IO requirements

At one of the clouds we used a non-parallel distributed filesystem and the developer experience was absolutely terrible. While dealing with large files was acceptable, the small files experience was extremely slow - it'd take 30 minutes to install a basic Conda environment and 2 minutes to run `python -c "import torch"`. This is because Python has tens of thousands of 4-16KB files and if the file system isn't optimized to handle those and the meta-data servers are weak, it'd be a very frustrating experience.

In general a typical Python shop needs a filesystem that can deal with:

- tens of thousands of tiny files
- few huge files

But, of course, only you know what your workloads' specific requirements are. Also consider the relationship between local storage and remote (shared) storage, as some providers will reduce the size and performance of local drives to save money. In many cases, developers will read data from a shared filesystem that can be cached locally (code libraries, models, datasets). Teaching people how to use rsync with local NVMe can improve the developer experience, and reduce I/O on the shared filesystem.

Please refer to the notes and guidance in the Storage chapter to know the nuances of storage requirements and their benchmarking.

What happens when storage fails

With advanced expensive distributed filesystems the chance of failure is relatively small, but it's quite big with cheaper storage solutions.

But it may still happen with any system.

You need to know:

- Who is in charge of fixing the problem?
- How long will it take to recover?
- Who pays for the downtime?
- What are the users to do while there is the problem?

If the resolution will take a long time often one needs to add another temporary filesystem partition to enable people to do their work. And, of course, you will have to pay for it.

Region migration

A cluster may be forced to migrate to a different region when upgrading to a next generation accelerators or expanding the capacity, if the region you're in doesn't have what you need. The storage has to be in the same region as the accelerators for the workflows to be fast.

The migration event triggers a sometimes very painful storage migration experience.

Here are some critical questions you need to ask long before the migration starts.

- Is the provider responsible for moving your data or is it your responsibility?
- Have you checked that the provided tooling is good enough to move TBs of data in a few hours, or will it takes many days to move? For example, using a storage cloud to migrate will typically drop all file metadata, which can be a huge problem. If you have 5 million tiny files, it could take forever to copy. Unless you use `tar`, but which may take many hours to create and do you have the 2x storage to have 2 copies of your data?
- Are you supposed to pay for the storage and the compute for both overlapping clusters?
- What happens to the files being edited and created while the filesystem is on the move - do you send everybody home while the migration is happening and freeze the filesystem?

Backup and archive

Many CSPs only have one tier of file storage available at one price point. However, organiations can have needs for multiple tiers of storage. For example, you might want to archive old model checkpoints or finetuning datasets to cheap, cold storage such as S3 object on HDD.

Having the flexibility to expand your total storage capacity, and keep the "hot" (local NVMe), "warm" (shared NVMe), "cold" (shared HDD), and "archive" (tape) in sync can help improve the resiliency of systems, save money, and allow for easier migration or expansion over time.

Network

This segment is mostly relevant to those planning to do training and finetuning. If you need to rent accelerators either for inference via large deployments of microservices or for small, on-demand, interactive work (i.e. notebooks) you can safely ignore this information. The only exception is when you plan on inferencing very big models that require more than one node for a single replica.

In general you want to ensure that the offered intra-node and inter-node network speeds match the promise and your expectations. When you measure, ask which NCCL env vars the platform needs; a safe default is `NCCL_NVLS_ENABLE=2` (use NVLink SHARP when available).

Ask for the actual performance numbers

Compute theory never matches reality, and the reality may dramatically vary from provider to provider even if they all use the same components, as it depends on the quality of all involved components and how well the racks were designed and put together.

The easiest ask is to request an `all-reduce` benchmark plot over 4-8-16-32-64 nodes (or more if your cluster is more than 64 nodes). You'd expect the bandwidth to gradually become worse with more participating nodes, but not dramatically so. Some networks become very inefficient at higher number of nodes.

Please refer to Real network throughput for more details.

Ideally you want to benchmark at least a few payloads - the ones that are of a particular interest to you because you know that this is the collective payload you will be using in your workloads. I usually just start by asking for a plot of a big payload of about 4-16GiB (16GiB would get the best bandwidth on the latest fastest inter-node networks), which immediately tells me if the network is good. But it's very likely you will want to know 256-512MiB payload as well, so just as well ask for the wider range.

The most practical approach I've used is to ask the CSP for their own `nccl-tests` `all-reduce` numbers - on 1 node, and on 4-16 nodes. Every provider wants to put their best foot forward, so what comes back is their best case, which is exactly what you want: it becomes the number you hold them to in your SLA. You can ask around whether those figures look right for that hardware, and in my experience they usually do. Then once you have the nodes, run the same benchmark yourself - those are the numbers you should be able to reproduce. If you can't, something about your allocation differs from what they demoed, and now you have a specific, quantified thing to escalate instead of a vague complaint that the network feels

slow.

footnote: of course, ask them about this upfront, not when you negotiate the SLA. That's too late and they then have no incentive to show you what their best case looks like.

I'm yet to see NVLink not performing the same everywhere (unless it's misconfigured on the software level), but still ask to also see the single-node benchmark.

Does the network steal from the accelerator memory?

One surprise I experienced on one of the clouds is that when I started using the GPUs I discovered that 5GiB of each was already used by the networking software - we managed to reduce it to a lower value, but still we were sold GPUs with less than their memory size and nobody told us about that before we signed the contract.

As accelerators become much bigger this will probably become unimportant, but when you get 75GiB of usable memory instead of 80GiB on H100 - that's a huge amount of memory lost per GPU.

InfiniBand or Ethernet?

In general, CSPs follow NVIDIA's DGX SuperPOD Reference Architecture which provides a lot of detail on how to build a rail-optimized InfiniBand network. Rail-optimized basically means that each GPU in an 8-way system connects to it's own leaf switch. Everything else is a standard fat-tree.

However, many of the largest GPU clusters in the world now run RoCEv2 instead of InfiniBand. Meta has proven that you can train frontier-class Llama models on a RoCEv2 network. Semianalysis/Fabricated Knowledge show a significant drop-off in NVIDIA's networking attach rate for their GPUs.

Since multi-node training depends on network collectives (i.e. NCCL or RCCL), the type of network can significantly impact performance and user experience.

How to evaluate the cluster

Once you have trial nodes, measure the three subsystems the rest of this article is about. Do not take the CSP's slide deck as the SLA until you can reproduce it on *your* allocation. The tools live in this book; an agent runbook for the same loop is evaluate-cluster/SKILL.md.

Use an **isolated venv**. Do not mutate the node's `dev` conda/env. If a recent `torch` with a CUDA build that matches the GPU is already there, reuse it; otherwise install the current stable CUDA wheel, plus `nvidia-ml-py` (MAMF telemetry), `matplotlib` (all-reduce plots), and `fio`.

Before expensive sweeps, run `torch-distributed-gpu-test.py` on one node so a broken NCCL/GPU shows up in seconds rather than inside a 16 GiB all-reduce.

A. Network

Intra-node: `all_reduce_bench.py` with `torchrun --nproc_per_node=8` (or however many GPUs the node has). Inter-node: the same script on **at least 4 nodes**. Passwordless SSH between those nodes is required for the `pdsh` recipe in the script header; SLURM `srun` is the other supported launcher. See Networking Benchmarks and Real network throughput.

The number to report is **`busbw`** (unidirectional). Intra-node: compare to advertised NVLink (not duplex); NVLS/SHARP can push the ring-formula number past 100% of that spec and only helps **all-reduce**. Inter-node: quote `busbw` and the NCCL path; **do not** divide by advertised IB/RoCE/EFA GBps — that collective is never a NIC-only measurement. If you only have one node, skip inter-node and write that down; a 1-node NVLink number does not tell you how the fabric behaves at 4 nodes.

B. Compute

MAMF sequential, siblings idle. `mamf-finder.py --search auto` on **each GPU** of the node, one at a time (`CUDA_VISIBLE_DEVICES=i`). That is the lemon-detection pass: one slow GPU ruins the node. Compare MAMF/MSMF to official TFLOPS in the accelerator tables. Details: benchmarks README.

DCGM (`dcgmi diag -r 2`). Hardware health (memory, PCIe/NVLink), not a FLOPS number (≲10.5 min on 8 GPUs). `-r 1` is software-only and not enough; `-r 3` is optional soak. Needs `nv-hostengine` (NVIDIA CUDA repo package `datacenter-gpu-manager-4-cudaN`). If the container cannot run it, write that down and keep MAMF.

A second *concurrent* all-8 run is a different question (shared power/cooling pulls the saturated clock down). Useful, but it is not the headline for "are these GPUs healthy."

C. Storage

`fio-scan` on the **local NVMe** path *and* the **shared filesystem** path, as documented under fio. You need both: NVMe is the ceiling; shared FS is what dataloaders and checkpoints actually hit. Default is 6 benches × 3 minutes per mount (~18 min each). Tiny-file (16 KiB) latency is the developer-experience proxy; 1 GiB sequential is the checkpoint proxy.

Write a **Conclusion** at the end of the dated report: one verdict per of compute / network / storage, then a **Flags** list (Gaps and anything that is merely “usable, not great”). Do not make the reader hunt through tables for the call.

Security

Though it can sometimes be an afterthought, CSP’s approach to security can vary widely. Just achieving a SOC 2 Type 2 compliance certification may not be enough. It is a good idea to check if the machines you’ll be using are virtualized. If you’re not in a VM, and the cloud provider serves other tenants, you may not trust what they are doing on the machines that you aren’t on. It’s a good idea to check that your cloud provider is verifying known-good versions of BMC firmware, system and BIOS firmware before provisioning (or re-provisioning) a server for you to use.

Miscellaneous

Tier 1 vs Tier 2 clouds

I don’t yet have a clear recommendation for whether Tier 1 clouds (AWS, GCP, Azure, etc.) vs emerging smaller Tier 2 clouds are better. My intuition is that Tier 2 clouds are likely to provide a better and more personal support as they have to work harder to secure customers.

Price-wise, Tier 2 clouds in general are cheaper because otherwise they won’t be able to compete with Tier 1 clouds. However, it’s obvious that their "margin" will be much smaller, because Tier 2 clouds don’t have the volume buying power of Tier 1 clouds.

Tier 2 clouds are more likely to be more flexible, have non-mainstream accelerators (e.g., AMD and Intel) and probably are more likely to lend hand at tuning things up at no to little cost.

Orchestration

A well-oiled node orchestration is critical for successfully using multi-node clusters.

Make sure you know which one you need - usually SLURM, Kubernetes or a combination of the two and make sure it’s well supported. Some clouds would only support one of them, or provide a very limited support for another type. These days SLURM is mostly used for training/finetuning and Kubernetes for inference. And there are other emerging orchestration platforms out there.

Same as with hardware, depending on whether you’re planning to administrate your own cluster you need to know who

will deal with any problems. This is a very crucial component of your stack, since if the orchestration is broken, nobody can use the cluster and you lose time/money.

Up-to-date software/OS versions

Make sure to ask that the provider isn't going to force you into some old versions of the software and an operating system.

I have had experiences where we were forced to use some very old Ubuntu versions because the provider's software stack which we had to use wasn't supporting more recent and up-to-date OS.

System administration

These days it can be difficult to find a good system administrator that understands the specific needs of the ML workloads, so it's a good idea to ask if some of that work could be offloaded to the CSP. Tier-1 CSPs sub-contract service companies that can provide various degrees of system administration. Smaller clouds are likely to offer their own direct services. They usually have a good grasp of what ML workloads need.

You won't be able to succeed without someone experienced taking care of your cluster. Using your ML engineers to also deal with system administration work can be very counter-productive, since it can be a very time-demanding and interrupting work.

Either hire a system administrator or hire a service company that will do it for you.

Conclusion

These notes are based on my direct experience and clearly I haven't been exposed to all possible things that may go wrong and wreck havoc with your cluster or make your whole team burn out and lose a lot of their hair. But this should be a good foundation to start thinking about.

Add your own questions, by thinking what's important for you, what failures may prevent you from accomplishing your compute goals.

If you have a particular CSP that you're casing out ask the community about them, especially what pitfalls to avoid with that cloud.

The key message of this article is for you to choose a cloud where your choice hasn't been taken away and that you don't get stuck with a service your developers hate, which is likely to lead to people leaving your company.

If you feel that these notes are overwhelming for you, I occasionally consult helping with due diligence and joining discovery calls. You can contact me at stas@stason.org.

Additional reading

- semianalysis.com created a ClusterMax CSP rating system and includes excellent explanations of the different criteria and plans to continue ranking many CSPs. 2025

When Is It Worth Upgrading GPUs?

A practical guide to deciding whether a GPU generation upgrade is worth its cost, using a real DeepSpeed training benchmark (meta-llama/Llama-3.1-8B, FlashAttention-3 on H200 vs FlashAttention-4 on B200) as the concrete worked example. The framework generalizes to any "should I upgrade from GPU X to GPU Y" decision building upon numbers from Hopper H200 → Blackwell B200.

Key spec differences: H200 vs B200

Spec	H200 SXM	B200 SXM	Ratio (B200/H200)
Architecture	Hopper	Blackwell	—
Compute capability	sm90	sm100	—
Attention kernel (HF auto-select)	FlashAttention-3	FlashAttention-4 (beta)	—
Peak bf16 TFLOPS (theoretical)	989	2250	**2.28×**
Peak fp8 TFLOPS (theoretical)	1979	4500	2.27×
HBM capacity (usable) HBM3e	~141GiB	~180GiB	1.28×
HBM bandwidth	4.8TBps	8.0TBps	1.67×
TDP	700W	1000W	1.43×

Theoretical peak TFLOPS is a ceiling nobody hits in practice — see the MAMF numbers below for what's actually achievable. Full spec tables (all dtypes, all vendors, memory, power, clocks): ml-engineering accelerator spec tables.

MAMF: the realistic ceiling, not the marketing number

MAMF (Maximum Achievable Matmul FLOPS) is the actual best-case `matmul` throughput measured on real hardware/software (perfectly-aligned max-size shapes, no sparsity) — as opposed to the theoretical peak that's physically unreachable. Use it to sanity-check whether your own training TFLOPS have room left to optimize, or whether you're already near the ceiling (measure MAMF on your own GPUs with `mamf-finder.py`). Source and full comparison across multiple vendors: MAMF comparison table.

BF16:

Accelerator	MAMF (TFLOPS)	Theoretical	Efficiency
H100/H200 SXM	794.5	989	80.3%
B200 SXM	1745.0	2250	77.6%

FP8:

Accelerator	MAMF (TFLOPS)	Theoretical	Efficiency
H200 SXM	1453.4	1979	73.4%
B200 SXM	3432.5	4500	76.3%

Takeaway: the *achievable* bf16 ratio (1745/794.5 = **2.20×**) tracks the theoretical 2.28× closely because efficiency is similar on both chips (80.3% vs 77.6%). In fp8 the achievable ratio is actually *higher* than theoretical (3432.5/1453.4 = **2.36×**) because B200's fp8 path is relatively more mature (76.3%) than H200's (73.4%). Either way, **2.2–2.4× is the realistic matmul-only ceiling** for this upgrade — real training throughput will land below that once attention scaling, dense-GEMM maturity, and communication overhead are factored in (the worked example below shows exactly how much below, and why).

Decision framework, part 1 — do you need a new dtype?

Start here, because this one question can settle the decision on its own, with no benchmarking. The real first question isn't *"how much faster is it"* but *"does the new GPU do something the current one **can't**?"* — and for a hardware-generation jump the clearest such capability gap is new low-precision dtypes.

Blackwell adds **FP4/NVFP4/FP6** that Hopper **cannot execute in hardware at all** — relevant for FP4 inference or microscaling training recipes.[^unlocks] If your target recipe needs one of these formats, the old GPU scores zero, not "2× slower" — there's nothing to benchmark, you either need the capability or you don't. (Caveat: this is also the feature *most* exposed to software-maturity risk — the silicon can do it before the framework/kernels/numerics recipes can; see software support below.)

Before upgrading *for* a feature, confirm the software you actually use supports it. A hardware capability you can't exercise from your stack is money spent on nothing — a spec sheet lists what the silicon *can* do, not what your framework/kernels/libraries will let you do today. Examples:

- **NVLink-C2C** (the CPU↔GPU coherent interconnect on Grace-based systems) looked great on paper but was barely exploited by most training/inference software when it launched — so paying for it bought little real-world benefit until the software caught up.
- Similarly, it took many months for a fully usable flash attention 4 to emerge. One had to continue using flash attention 2, which was very slow on Blackwell, since it couldn't take advantage of the new architecture.

Treat "does my stack support this feature?" as a hard gate on any feature-motivated upgrade (and re-check over time, since support improves).

Everything else a new GPU brings — more memory, more bandwidth, faster interconnect, more raw FLOPS — is a *bigger quantity of what you already have*, not a brand-new capability, so it doesn't settle the decision by itself; it comes down to measurement. That's part 2.

[^unlocks]: Dtype peak TFLOPS (theoretical), H200 → B200: bf16 989→2250, fp8 1979→4500, plus formats Hopper can't execute at all — fp6 →4500, fp4 →9000, nvfp4 →10000. FP4 roughly doubles fp8 throughput and halves the memory/bandwidth of the quantized tensors. What makes 4-bit actually *usable* is Blackwell's **2nd-gen Transformer Engine with microscaling**: FP4 has only ~16 representable values, so one scale factor per tensor can't span its dynamic range (small values flush to zero, large ones saturate). Microscaling (MX / NVFP4) instead uses a separate scale per small block (e.g. per 16–32 values), applied by the tensor cores in hardware at no throughput cost — so you keep accuracy near fp8/bf16 at FP4's ~2× speed and half memory, which naive per-tensor FP4 can't. (Hopper's 1st-gen TE did this for fp8 via per-tensor scales; Blackwell's 2nd-gen extends it to block-scaled fp4/fp6.)

Decision framework, part 2 — is it worth it on performance?

If you don't need the new functionalities covered by part 1, then upgrading is a quantitative speed-and-capacity-for-money question — and that you have to **measure.** Work through these steps before trusting any vendor slide or single benchmark number:

1. **Match the software stack on both GPUs.** Same torch/CUDA/library versions on old and new hardware — a newer GPU is often *also* validated on a newer stack, and stack deltas alone can be worth double-digit percent (see "compare on an identical software stack" below). Compare apples to apples.
2. **Find your workload's attention-vs-dense split, and how it scales with your real config.** New GPU generations usually improve attention (FlashAttention kernels) faster than dense GEMM maturity (cuBLAS/cuDNN need time to catch up on a new architecture). If your workload is attention-heavy (long context), you'll see a bigger win than a short-context/dense-bound workload — and that split isn't fixed, it grows with sequence length. Measure it, don't assume it.[^linear-attn]
3. **Weigh the new GPU's quantitative hardware advantages for *your* workload — memory, bandwidth, interconnect.** These are lifts, not new capabilities, so their value is workload-specific and must be measured:
    - **More memory** (e.g. 180 vs 141GiB) is the borderline one. It can let a config fit that OOMs on the old GPU and lets you shed throughput-costing crutches (CPU offload, a high ZeRO stage, activation checkpointing) — so the *effective* speedup can even exceed the raw FLOPS ratio once you re-tune. But "doesn't fit" is rarely a hard wall: you can almost always fit on the older GPU with more parallelism (more GPUs + ZeRO/TP/CP/SP/PP/EP), so it's usually a *cost/efficiency* trade-off, tipping to "just upgrade" only when the extra GPUs/parallelism to fit would cost or slow you more than the newer hardware. (In the extreme — when no amount of parallelism you can afford fits it — memory does become a hard capability gap, like a new dtype.)
    - **More HBM bandwidth** (8.0 vs 4.8TBps, ~1.67×) sets the ceiling for bandwidth-bound work (norms, elementwise, optimizer, KV traffic) and decode-bound inference latency; payoff depends on how bandwidth-bound your workload actually is.
    - **Faster interconnect** (newer NVLink/NVSwitch/NVLink-C2C) raises the ZeRO/TP/CP/SP/PP/EP comms, various host memory offloading strategies and multi-node-scaling ceiling; payoff depends on how comms-bound you are.
4. **Compute $/token (or $/step), not just TFLOPS/step.** A 2× speedup at 2× the price is break-even on cost, with the tie broken by (a) wall-clock time and (b) memory headroom.
5. **Watch out for real-world caveats that inflate or deflate the lab number** — packed vs unpacked sequences, beta-quality kernels, software-stack mismatches. See the Caveats below for concrete examples.

[^linear-attn]: The $O(s^2)$ full-attention share that drives most of this long-context advantage is shrinking as newer models move to **linear / hybrid attention** (Gated DeltaNet, Mamba-style SSMs, sliding-window, and MoE hybrids that interleave a few full-attention layers among many linear-attention ones). Linear attention is $O(s)$, so it behaves more like the dense path than like full attention — the quadratic term that makes the new GPU shine at long context is a *smaller fraction* of these models than of a classic dense-attention transformer. If your target model is a hybrid/linear-attention architecture, don't extrapolate the speedup from a full-attention model like the Llama-3 model family; benchmark that specific architecture (its kernels are also "younger" — see software support below).

The rest of this doc walks through these steps end-to-end on the H200→B200 example.

Worked example: meta-llama/Llama-3.1-8B training step, FA3 vs FA4

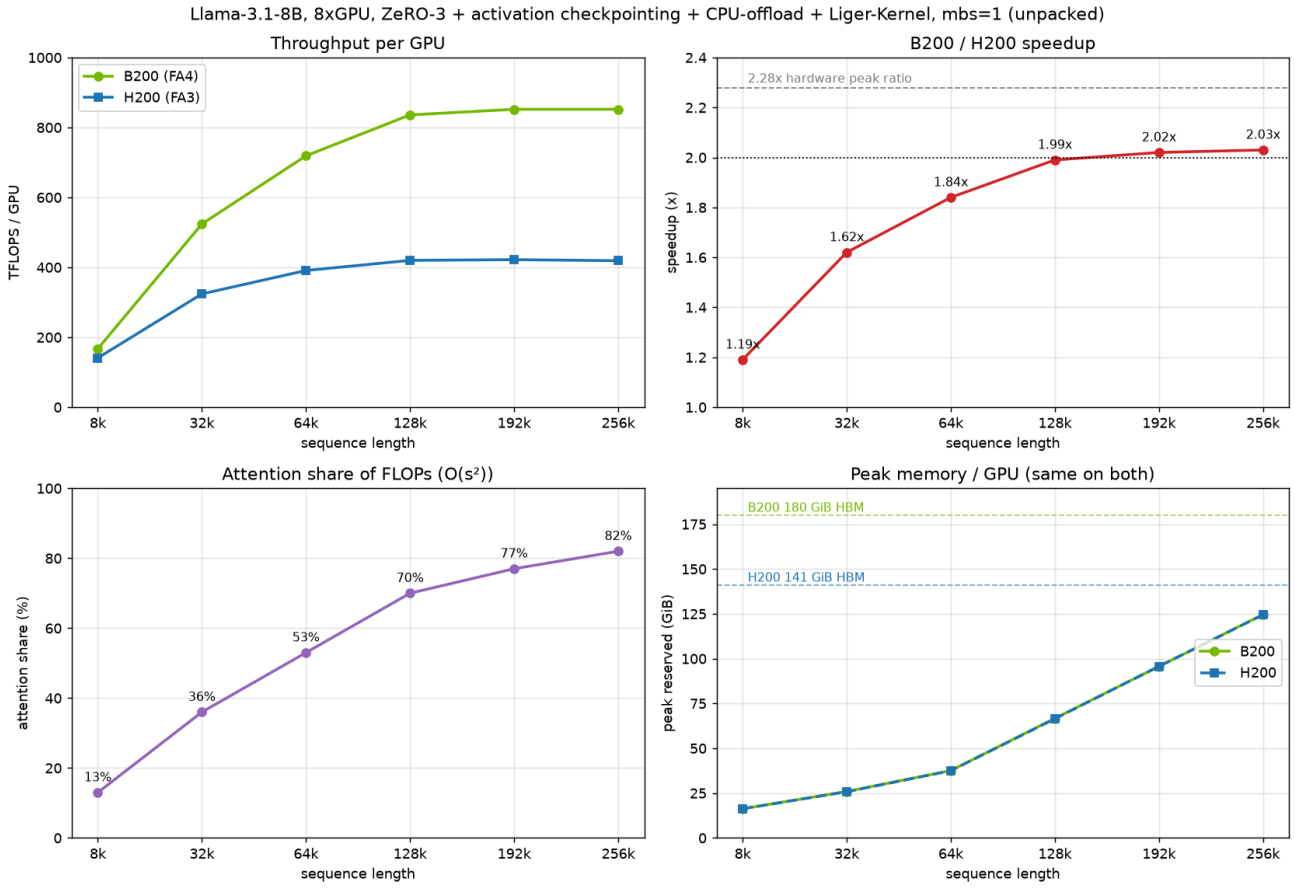
Setup: meta-llama/Llama-3.1-8B, real weights, bf16, 8 GPUs, DeepSpeed, mbs=1, fwd+bwd+step. Attention is HF-native

`attn_implementation`, auto-selected per GPU: **H200 → FA3, B200 → FA4.**

Results: speedup grows with sequence length

Same config on both GPUs (ZeRO-3 + activation checkpointing + CPU-offload + Liger-Kernel), steady-state TFLOPS/GPU:

seqlen	full attn share of step	H200 TFLOPS	H200 MFU	B200 TFLOPS	B200 MFU	speedup
8K	~13%	141	14%	167	7%	**1.19×**
32K	~36%	324	33%	524	23%	**1.62×**
64K	~53%	391	40%	719	32%	**1.84×**
128K	~70%	420	42%	836	37%	**1.99×**
192K	~77%	422	43%	852	38%	**2.02×**
256K	~82%	419	42%	852	38%	**2.03×**



Why speedup scales with seqlen (the attention-vs-dense split, in action): causal attention costs `6·s²·d·h·L` per step (O(s²)); the dense path costs `6·N·tok` (O(s)). Longer sequences push attention's share of the step up, and attention is exactly where FA4/Blackwell's advantage concentrates — while the dense path (bandwidth-bound at this size + "young" sm_100 cuBLAS) that scales only ~1.7× shrinks to a rounding error. The identity `speedup ≈ 2.28 × MFU_ratio` holds throughout — the residual gap to 2.28× is FA4-beta overhead not yet reaching the full hardware ratio even on pure attention.

FLOP-counting convention: the TFLOPS/MFU figures are *model* FLOPs (MFU) with attention counted **causally** ($6 \cdot s^2 \cdot \ldots$, the lower triangle FlashAttention actually computes — the non-causal $12 \cdot s^2$ would double-count attention and inflate MFU at long context). Activation-checkpoint recompute is *excluded* — counting it would be HFU (~1.3× higher). Both conventions cancel in the B200/H200 speedup ratio, so only the absolute MFU/TFLOPS depend on this choice, not the headline speedups.

8k caveat: this heavy config (ZeRO-3 + activation checkpointing + CPU-offload) exists to reach 256k, so at 8k it's pure overhead with no compute to hide behind → single-digit B200 MFU and only 1.19×. A right-sized 8k run (ZeRO-2, no activation checkpointing/offload) hits ~1.69× at much higher MFU — see "where the dense-path deficit comes from" below. **Lesson: don't benchmark a long-context config at short context and call it representative.**

Does the bigger GPU unlock configs the old one can't run?

Config: ZeRO-3, Liger-Kernel, activation checkpointing on, CPU AdamW offload on, mbs=1. With activation checkpointing + fused-CE, activation memory is small and linear: **peak ≈ 12.6GiB + 0.00043GiB/token** (the 12.6GiB floor is ZeRO-3 param/grad shards; AdamW state lives on CPU).

Peak memory (`max_memory_reserved`, worst rank) is **identical on H200 and B200 at a given seqlen** — it's set by the model + ZeRO-3 sharding + activations, not the GPU architecture — so a seqlen that fits one fits the other, up to each GPU's own ceiling:

seqlen	H200 peak (GiB)	B200 peak (GiB)	H200 TFLOPS	B200 TFLOPS
8K	16.3	16.3	141	167
32K	25.8	25.9	324	524
64K	37.5	37.6	391	719
128K	66.5	66.7	420	836
192K	95.7	95.7	422	852
256K	**124.5**	**124.6**	**419**	**852**

Max on H200's ~130GiB usable budget is **256k tokens at ~125GiB**, confirmed on both boxes (peaks match within ~0.2GiB). B200's larger HBM (180 vs 141GiB) could push past ~350k tokens — sequences H200 **cannot fit at any price** — and that's exactly the regime where B200's speedup is largest. This is the "more memory" advantage from part 2 in its extreme form: past 256k the comparison isn't "B200 is faster", it's "H200 can't do this job".

Caveats

1. unpacked vs packed sequences

   Every run above feeds **one dense sequence** → genuine full $O(s^2)$ attention, which is what drives attention to ~82% of FLOPs at 256K (the "full attn share of step" column of the speedup table) and the ≥2× speedup. Real training often **packs** short samples with a document mask, making attention cost $\Sigma\ \text{len}_i^2$ instead of $(\Sigma\ \text{len}_i)^2$. For example, packing samples of ~8k tokens each into a 256k sequence → attention runs only *within* each sample, so its FLOPs drop ~32× (total attention work falls from 256k^2 to $\Sigma\ \text{len}_i^2 = 256\text{k} \cdot 8\text{k}$, a factor of $256\text{k} \div 8\text{k} = 32$; the drop factor = packed seqlen ÷ *average sample length*, set by how **short** each sample is, not by how **many** you pack). That puts attention back at the ~13% share of a genuine 8k sequence (the 8K row above), so the speedup reverts to the dense-bound regime we measured directly there — **~1.7×** (see "where the dense-path deficit comes from" below), not the ~2× of true long context. The ≥2× result is specific to genuinely long *individual* documents — check which regime your workload is

actually in before assuming the headline number transfers.

2. compare on an identical software stack

   When I was writing this guide I originally didn't pay attention and compared H200 on torch **2.10.0/cu129** against B200 on torch **2.13.0/cu130** and got very different results, massively benefitting B200. Suspecting the CUDA version, I reran the sweep on H200 alone across torch 2.10.0–2.13.0 on both cu129 and cu130 — and CUDA turned out to barely matter. The real driver is **PyTorch itself:** 2.12.0 and higher is significantly faster than 2.11.0 and earlier, independent of CUDA version. At 8k, same GPU/model/config, sweeping torch and CUDA independently:

H200 @ 8k, TFLOPS/GPU	torch 2.10.0	2.11.0	2.12.0	2.13.0
cu129	113.8	–	–	140.4
cu130	117.8	120.5	140.1	**140.2**

   Full-range torch 2.10.0 → 2.13.0 on the H200 is **+19% at fixed cu130** (117.8 → 140.2TFLOPS), and almost all of that gain lands in one place — the torch 2.11.0 → 2.12.0 jump alone is **+16%** (120.5 → 140.1). CUDA, meanwhile, contributes at most ~3.5% (113.8 → 117.8 at torch 2.10.0) and essentially nothing by torch 2.13.0 (140.4 vs 140.2). So the original 2.10.0/cu129 → 2.13.0/cu130 comparison that kicked off this investigation (113.8 → 140.4, **+23.4%**) was almost entirely a **PyTorch-version artifact, not hardware, and not even CUDA.** All results in this doc use the identical stack (torch 2.13.0/cu130, deepspeed 0.19.2) on both boxes. **If you skip this check, you will systematically overstate whichever GPU happens to be benchmarked on the newer PyTorch version.**

Is B200 worth ~2× the H200 cost?

For simplicity let's assume the cost difference is 2×, which is usually the case when a new GPU family just starts to come out. Blackwell had been out since mid-2026, so the actual cost difference is already less than 2×.

In your actual calculations use your quoted price, not the information found online.

So let's do the math with the 2× price difference assumption:

- **Long unpacked context (128k–256k):** ~2.0× throughput for ~2× cost → break-even on $/token, then a net win from (1) **half the wall-clock time** and (2) **more memory** unlocking context (≥256k) H200 can't hold at all — and that's exactly where B200's speedup peaks, so the two compound.
- **Short / packed-short (dense-bound):** ~1.7× for 2× cost → **loses** on $/token today. At this batch size the dense path is bandwidth-bound, not compute-bound (see "where the dense-path deficit comes from" below), so don't expect cuBLAS sm_100 maturity alone to close the gap to 2.28× — it converges toward the **~1.67× HBM-bandwidth ceiling** instead. Reaching closer to 2.28× here would need a bigger batch/larger GEMMs to make the dense path compute-bound again.

Beyond $/token — the time dimension. Cost-per-performance alone can be the wrong metric, because it treats a slow-but-cheap and a fast-but-pricey option as equivalent when they cost the same per token — ignoring *wall-clock time to result*.

- For training, a run that takes 2× longer can mean losing a first-to-market race, which no $/token saving compensates for.
- For inference, 2× slower performance directly degrades user experience — higher TTFT (time to first token) and TPOT (time per output token) — which can cost you users regardless of how favorable the per-token math looks.
- In principle you can buy back the speed on the cheaper GPU by throwing ~2× as many of them at the problem, but that usually means *more* cross-GPU (and often cross-node) communication, which adds

overhead and can eat into — or even exceed — the speedup you were trying to recover, especially for comms-bound workloads. So weigh the raw speed advantage as its own axis, not just as a term inside the $/token ratio.

Verdict for this example: yes, upgrade for genuinely long-context training; hold off for short/packed workloads, since the dense path there is bandwidth-bound rather than compute-bound (see "where the dense-path deficit comes from" below) and won't reach the full 2.28× hardware ratio at this batch size.

Where the dense-path deficit comes from (why not the full 2.28×?)

Deep-dive on the 8k step (ZeRO-2, no activation checkpointing, right-sized for short context) explaining *why* the dense path caps the speedup below the hardware ratio. All three buckets below share **one clock** — the measured wall-clock step time, split by each bucket's share of compute-stream kernel time (attention = FlashAttention kernels; NCCL gradient comms overlap the backward pass and are hidden inside the wall clock) — so the buckets reconcile with the total by construction. Reproduce with `bench_decompose.py`.

Step decomposition @ 8k (attention is ~13% of the FLOPs and ~12% of the step time; the dense path is the rest):

bucket	H200 TFLOPS	H200 MFU	B200 TFLOPS	B200 MFU	ratio
attention (FA)	447	45%	834	37%	1.87×
everything else	463	47%	771	34%	1.67×
total	461	47%	778	35%	**1.69×**

Two things stand out:

- **Attention (FA4 vs FA3) scales 1.87×**, measured *in-step*, matching the isolated micro-benchmark (`bench_flash_attn.py`, FA4 vs FA3, 8B attn shape, causal: fwd+bwd **1.88× @ 8k → 1.95× @ 32k**). Attention is close to the 2.28× ceiling and is *not* the bottleneck.
- **The ceiling is the dense "everything else" bucket at 1.67×** — essentially exactly the **HBM-bandwidth ratio** (8 vs 4.8TBps = 1.67×). At mbs=1 × 8k the per-GPU GEMMs are small and the dense path is bandwidth-/overhead-bound (norm/RoPE/SwiGLU/CE elementwise + NCCL comms + skinny GEMMs), so Blackwell's 2.28× *compute* advantage sits mostly idle and the bucket tracks memory bandwidth instead. That drags the blended total down to **1.69×**.

Liger-Kernel (fused RMSNorm/RoPE/SwiGLU/CE) is a **memory win, not a speed win at 8k**: it cuts peak memory ~20% (**99 → 80GiB**, identical on both GPUs) while leaving step time within noise (≤1.5% on either box) and the blend essentially unchanged (~1.66×). Fusing the bandwidth-bound elementwise ops doesn't lift throughput here because the dense path is already limited by GEMM size and comms, not by those ops — consistent with the bucket tracking HBM bandwidth. cuBLAS sm_100 is "younger" than Hopper's cuBLAS and FA4 was in beta when these numbers were taken and has since shipped as the `flash-attn-4` package, so this dense-path deficit should shrink over time — re-run this benchmark periodically rather than treating today's number as permanent.

Environment used for these experiments

	B200	H200
GPU	NVIDIA B200	NVIDIA H200
attention	flash-attn-4 4.0.0b22 (beta)	flash-attn-3 3.0.0

	B200	H200
PyTorch / CUDA	2.13.0 / cu130	2.13.0 / cu130
deepspeed	0.19.2	0.19.2
python / transformers / Liger-Kernel	3.12.12 / 5.14.1 / 0.8.0	(same)

Software support: the cost of switching too early

New silicon almost always ships ahead of the software stack that can fully exploit it. "The GPU is available" and "the GPU is *usable* for your workload at its rated performance" can be many months apart — and the gap can be per-model-architecture, not just per-GPU. Two concrete forms this takes, both visible directly in this benchmark:

1. **The optimized kernel itself may still be beta.** B200's attention path here is `flash-attn-4 4.0.0b22` — a beta release, not GA. Beta kernels can have correctness bugs, missing features, and breaking API changes before their 1.0; pin the exact version, re-test on every bump, and check the project's issue tracker rather than assuming "it's on PyPI" means production-ready.
2. **The rest of the stack may lag even where the headline kernel is fine.** Isolated attention (FA4 vs FA3) already scales at ~1.9× here, close to the 2.28× hardware ratio (see "where the dense-path deficit comes from" above) — but the *dense* path only reaches ~1.7× (it tracks the HBM-bandwidth ratio, being bandwidth-/overhead-bound at this size, and Blackwell's cuBLAS on the new `sm_100` architecture is "younger" than Hopper's, which has had years to mature). Net effect: you can be running the best available attention kernel and still see a fraction of the theoretical speedup, because a less glamorous part of the stack — the one nobody puts on a slide — hasn't caught up yet.

Practical checklist before committing budget to a new architecture:

- Search the kernel library's issue tracker for your exact GPU's compute capability (e.g. `sm_100`) and your model's attention/MoE mechanism by name — a README saying "supported" doesn't mean GA-quality or even that pre-built wheels exist.
- Confirm your training framework (DeepSpeed, Megatron-LM, TorchTitan, etc.) has a *released* version validated against the new compute capability — "should work, it's just CUDA" is not the same as tested.
- If your model uses a non-standard op (linear attention, MoE routing, sparse attention, fused custom CUDA extensions), verify someone has actually built *and benchmarked* it on the new architecture. Don't assume it inherits the GPU's speedup just because the GPU is faster.
- Re-measure periodically rather than trusting a one-time verdict. Beta kernels and young GEMM libraries both improve fast; the cuBLAS-maturity gap is exactly the kind of moving target that can flip a "not yet worth it" into "worth it" within a few release cycles — in either direction if a regression lands.

Rent the new target hardware

While you might be able to rely on someone else doing the analysis work for you (this guide), typically workloads differ and it's best to benchmark your exact workloads.

Therefore, I recommend first renting at least one node of the target GPUs you're considering and running benchmarks there.

If your workload involves multiple nodes, try to first reduce the workload to just one node and do the performance comparisons there. e.g. shrink the number of layers in the model to make it much smaller — ML model architectures are typically a stack of identical repeating layers, so cutting a 48-layer model down to 1-2 layers consumes little GPU memory and should fit into a single node easily, while keeping the per-layer compute/memory behavior representative.

If the workload involves multiple nodes, then you definitely need to eventually benchmark the full minimal replica

configuration, but you can postpone that until after checking that the single-node comparisons give favorable results towards choosing to upgrade the GPUs. If they do not, then you saved yourself time and money not needing to rent additional nodes.

Applying this framework to your own upgrade decision

Swap in your own model, GPUs, and prices, then work through the two-part decision framework — part 1 then part 2 — first the feature check, then, if features don't already decide it, the performance steps:

1. Pin an identical software stack on both GPUs (see "compare on an identical software stack" above) — otherwise you're measuring the stack, not the hardware.
2. Sweep the axis that shifts your attention/dense (or otherwise-accelerated/non-accelerated) split — for LLM training that's sequence length; for other workloads it may be batch size, resolution, or something else entirely.
3. Sweep memory-limited configs on the old GPU until it OOMs, and check what the new GPU can additionally fit.
4. Get your actual $/GPU-hour on old vs new, and compute $/token or $/step at each point on the sweep, not just at the top of the sweep.
5. Check for gotchas that invalidate the comparison: beta kernels, packed vs unpacked data, thermal throttling, silicon-lottery variance across GPUs on the same node (see Not all accelerators are created equal).

If you use FSDP for scaling — it should be somewhat equal in performance to Deepspeed ZeRO — you might have to tweak the config a bit to match your exact setup, but, of course, ideally benchmark your exact software.

Same-generation upgrades

Everything above compared two *generations* (Hopper → Blackwell). A within-generation refresh — e.g. B200 → **B300** ("Blackwell Ultra") — is a different, usually smaller, decision. The same two-part decision framework applies — part 1 (do you need a new dtype?) and part 2 (is it worth it on performance?) — but the answers shift in predictable ways.

spec	B200 SXM	B300 SXM	ratio
bf16 / fp8 TFLOPS (theoretical)	2250 / 4500	2250 / 4500	1.0×
fp4 / nvfp4 TFLOPS (theoretical)	9000 / 10000	12600 / 15000	~1.4–1.5×
bf16 MAMF (achievable)	1745	1769	~1.01×
HBM (usable)	~180GiB	~288GiB	**1.6×**
HBM bandwidth	8.0TBps	8.0TBps	1.0×
TDP	1000W	1300W	1.3×

What this means for the should-you-upgrade framework:

- Part 1 **(do you need a new dtype?) rarely fires.** B300 introduces no dtype B200 lacks. The one thing that *can* act like a part-1 gap is **memory**: 288 vs 180GiB (1.6×) is a big jump, so a model/config that can't be made to fit a B200 node even with parallelism may fit a B300 node outright.
- Part 2 **(is it worth it on performance?): the only real compute uplift is fp4 (~1.4–1.5×)** — bf16/fp8 are unchanged, so B300 mainly pays off for low-precision inference/training and for the extra memory.
- **Software-maturity risk is low.** Same architecture family (`sm_100`), same FA4/kernel stack — none of the

"switching too early" cost from the cross-generation case.

Reproduce this benchmark

Every number in this guide comes from the scripts in this folder. Nothing here is hand-entered — below is exactly which script produces which table/figure, so you can regenerate them (or re-run on your own GPUs/model). All scripts **auto-detect the GPU** (Hopper→FA3, Blackwell→FA4) and echo the resolved backend + full env in their output header, so the *same commands* run on both boxes.

Files at a glance

file	purpose	produces
`install.sh`	one-time setup: pins matched torch/CUDA/deps, installs FA3 (Hopper) or FA4 (Blackwell), stages the model to local disk	the identical-stack environment (the "compare on an identical software stack" caveat)
`benchmark.py`	training loop (fwd+bwd+step), real weights, prints per-step JSON (`tflops`, `peak_mem_gib`); config block at top, env-overridable (`SEQ_LEN`, `STEPS`, `WARMUP_STEPS`, `ZERO_STAGE`, `OFFLOAD_OPTIMIZER`, `GRAD_CHECKPOINT`, `USE_LIGER`, `MEM_DEBUG`)	one row of the speedup + memory tables
`run_sweep.sh`	drives `benchmark.py` across a list of sequence lengths, logs to `results_<gpu>_<ts>.log`	the **speedup-vs-seqlen** and **max-seqlen/ memory** tables
`bench_flash_attn.py`	isolated FA3/FA4 attention-kernel micro-benchmark (fwd + fwd/bwd TFLOPS across seqlens)	the **isolated attention** numbers in "where the dense-path deficit comes from"
`bench_decompose.py`	one profiled training step, splits measured wall time into attention (FlashAttention kernels) vs everything-else buckets on a single clock (env-overridable: `SEQ_LEN`, `ZERO_STAGE`, `GRAD_CHECKPOINT`, `OFFLOAD_OPTIMIZER`, `USE_LIGER`)	the **step-decomposition** table in "where the dense-path deficit comes from"
`plot_results.py`	matplotlib figure from the sweep numbers (data hardcoded at the top of the script)	`results_plot.png`

Step 1 — set up an identical stack on each GPU (once per node)

Run the same installer on the H200 box and the B200 box. It pins identical torch/CUDA/deps so the only intended difference is the attention backend, then auto-installs FA3 on Hopper / FA4 on Blackwell and stages the model to local disk:

```
./install.sh  # H200 -> FA3, B200 -> FA4 (auto-detected); --force fa3|fa4 to override
```

`HF_HOME` defaults to `~/.cache/huggingface`; the model (`meta-llama/Llama-3.1-8B`) must be present there first (e.g., run `hf download meta-llama/Llama-3.1-8B`). The installer's final step verifies the stack and does a tiny FA call to catch ABI

mismatches.

Step 2 — run the sequence-length sweep on each GPU (speedup + memory tables)

```
SEQLENS="8192 32768 65536 131072 196608 262144" GPUS=8 ./run_sweep.sh # -> results_<gpu>_<ts>.log
```

Run this on both boxes. Each step prints a JSON line like `{"step": 5, "warmup": false, "exec_time_ms": ..., "tflops": ..., "peak_mem_gib": ...}`; take the steady-state (non-warmup) `tflops` for the throughput/speedup table and `peak_mem_gib` for the memory table. The B200/H200 speedup is just the ratio of the two boxes' `tflops` at each seqlen. (Defaults if you omit env vars: single seqlen 196608, 8 GPUs, 10 steps + 2 warmup.) All the config that defines the run — ZeRO-3, activation checkpointing, CPU AdamW offload, Liger-Kernel, mbs=1 — lives in `benchmark.py`'s config block and is echoed in each run's header.

Step 3 — isolated attention micro-benchmark (dense-path deep-dive)

```
python bench_flash_attn.py   # prints per-seqlen fwd / fwd+bwd TFLOPS for FA3 or FA4
```

Run on both boxes; the FA4/FA3 ratio of `fwd_bwd_tflops` per seqlen is the "isolated attention" result showing attention itself scales ~1.9× — i.e. that the dense path, not FA, is what caps the end-to-end speedup.

Step 4 — step decomposition (attention vs everything-else buckets)

```
for L in 0 1; do USE_LIGER=$L deepspeed --num_gpus=8 bench_decompose.py; done  # L=0 base, L=1
Liger
```

Run on both boxes. It measures the wall-clock step time (profiler off), then profiles one window and splits that measured time into an attention bucket (FlashAttention kernels) and an everything-else bucket by their share of compute-stream kernel time (NCCL comm overlaps backward and is excluded as hidden). Because both buckets are carved out of the *same* measured step, they reconcile with the total by construction — that's the **step-decomposition** table. The B200/H200 ratio per bucket is the per-bucket speedup (attention ~1.87×, dense path ~1.67×).

Step 5 — the memory usage investigation (optional)

```
MEM_DEBUG=1 SEQ_LEN=32768 deepspeed --num_gpus=8 benchmark.py
```

`MEM_DEBUG=1` adds staged allocated/reserved probes around fwd/bwd/step and dumps a `torch.cuda.memory._record_memory_history` snapshot on rank 0 — this is useful for diagnosing GPU memory usage.

Step 6 — regenerate the plot

```
python plot_results.py       # -> results_plot.png
```

Note: `plot_results.py` has the sweep numbers **hardcoded at the top** (it does not parse the `.log` files). If you re-run the sweep and get different numbers, update the arrays in `plot_results.py` and the tables above to match, then regenerate.

Further reading

- Accelerators — full spec tables (all vendors, all dtypes), MAMF methodology, memory/power/clock references, silicon-lottery and cooling notes.

Compute

1. Accelerator - the work horses of ML - GPUs, TPUs, FPGAs, HPUs, QPUs, RDUs.
2. CPU - cpus, affinities
3. CPU Memory - how much CPU memory is enough - the shortest chapter ever.

Accelerators

Compute accelerators are the workhorses of the ML training. At the beginning there were just GPUs. But now there are also TPUs, FPGAs, HPUs, QPUs, RDUs and more are being invented.

There exist two main ML workloads - training and inference. There is also the finetuning workload which is usually the same as training, unless a much lighter LORA-style finetuning is performed. The latter requires significantly fewer resources and time than normal finetuning.

In language models during inference the generation is performed in a sequence - one token at a time. So it has to repeat the same `forward` call thousands of times one smallish `matmul` (matrix multiplication or GEMM) at a time. And this can be done on either an accelerator, like GPU, or some of the most recent CPUs, that can handle inference quite efficiently.

During training the whole sequence length is processed in one huge `matmul` operation. So if the sequence length is 4k long, the training of the same model will require a compute unit that can handle 4k times more operations than inference and do it fast. Accelerators excel at this task. In fact the larger the matrices they have to multiply, the more efficient the compute.

The other computational difference is that while both training and inference have to perform the same total amount of `matmuls` in the `forward` pass, in the `backward` pass, which is only done for training, an additional 2x times of `matmuls` is done to calculate the gradients with regards to inputs and weights. And an additional `forward` is performed if activations recomputation is used. Therefore the training process requires at 3-4x more `matmuls` than inference.

Subsections

General:

- Benchmarks

NVIDIA:

- Troubleshooting NVIDIA GPUs

AMD:

- Troubleshooting AMD GPUs
- AMD GPUs Performance

Bird's eye view on the high end accelerator reality

While this might be changing in the future, unlike the consumer GPU market, as of 2026-08 there aren't that many high end accelerators, and if you rent on the cloud, most providers will have more or less the same few accelerators to offer.

GPUs:

- As of today, ML clouds/HPCs already have B200s/B300s, and GB200/GB300 NVL72 racks are shipping. Rubin is expected to emerge in H2-2026.
- AMD's MI325X is now widely available on Tier 2 cloud providers. MI355X is starting to emerge. MI455X hopefully in 2026. New: large CSPs started to offer AMD GPUs
- Intel's Crescent Island is an inference-oriented Xe3P data-center GPU, with customer sampling in H2-2026. Intel has not published TFLOPS. See Intel Crescent Island

HPU:

- Intel's Gaudi2 and Gaudi3 are available at Intel's cloud.
- Falcon Shores is to replace Gaudi in 2025 - update - the project has been canceled

TPU:

- Google's TPUs are, of course, available but they aren't the most desirable accelerators because you can only rent them, and the software isn't quite easily convertible between GPUs and TPUs, and so many (most?) developers remain in the GPU land, since they don't want to be locked into a hardware which is a Google monopoly.
- Amazon's Trainium2 is very similar to the TPU architecture and is available on AWS

On Pods and racks:

- Cerebras' WaferScale Engine (WSE)
- SambaNova's SambaRack (SN40L today, SN50 from H2-2026)
- dozens of different pod and rack configs that compose the aforementioned GPUs with super-fast interconnects.

That's about it as of 2026-07-31.

The rest of this document will compare most of the above in details and if you want to read the specs please head here.

As most of us rent the compute, and we never see what it looks like, here is how an 8xH100 node looks like physically (this is the GPU tray of the Dell PowerEdge XE9680 Rack Server):

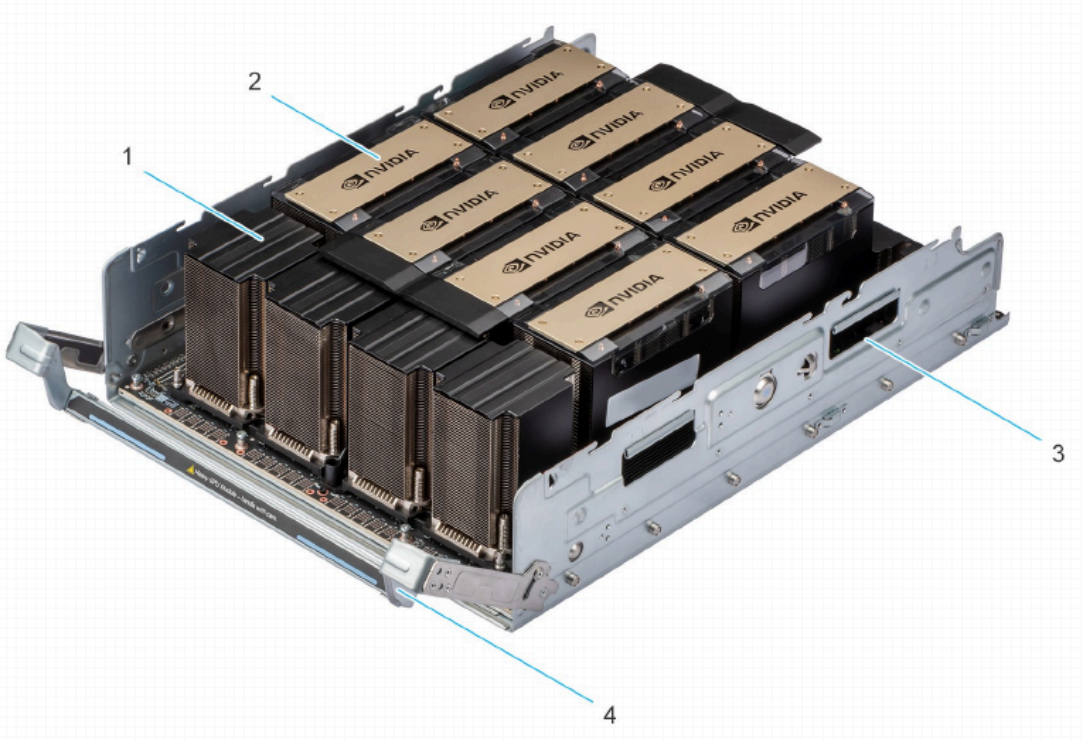


Figure 8. GPU H100/H800 Inside view

1. NVSwitch Heatsink
2. GPU Heatsink
3. Chassis
4. Handle

Glossary

- BLAS: Basic Linear Algebra Subprograms
- CPU: Central Processing Unit
- CTS: Custom Thermal Solution - a vendor-supported cooling option that allows a higher power limit than the default

- CU: Compute Unit - AMD's counterpart to NVIDIA's SM
- CUDA: Compute Unified Device Architecture - NVIDIA's GPU programming platform
- DCORE: the die core Intel reports Gaudi3 cache against; a Gaudi3 has 4 of them
- DGX: NVIDIA's turn-key accelerator server line
- ECC: Error Correcting Code
- FMA: Fused Multiply Add
- FPGA: Field Programmable Gate Arrays
- GA: Generally Available - the product can actually be bought or rented, as opposed to announced, sampling, or spec'd only
- GCD: Graphics Compute Die
- GEMM: General Matrix Multiply
- GPU: Graphics Processing Unit
- HBM: High Bandwidth Memory
- HFU: Hardware FLOPS Utilization
- HGX: NVIDIA's accelerator baseboard that OEMs build their own servers around
- HPC: High-performance Computing
- HPU: Habana Gaudi AI Processor Unit
- LLC: Last Level Cache
- MAMF: Maximum Achievable Matmul FLOPS
- MFU: Model FLOPS Utilization
- MIG: Multi-Instance GPU - NVIDIA's partitioning of one GPU into isolated instances
- MME: Matrix Multiplication Engine
- MSMF: Maximum Sustainable Matmul FLOPS
- NVL72: an NVLink domain of 72 accelerators; likewise NVL8 and NVL36
- OAM: OCP Accelerator Module - the Open Compute Project's accelerator form factor
- PSU: Power Supply Unit
- PTX: Parallel Thread Execution - NVIDIA's intermediate GPU instruction set
- QPU: Quantum Processing Unit
- RDU: Reconfigurable Dataflow Unit
- SM: Streaming Multiprocessor - NVIDIA's counterpart to AMD's CU
- SRAM: Static Random Access Memory
- TBP: Typical Board Power or Total Board Power, depending on vendor
- TDP: Thermal Design Power or Thermal Design Parameter
- TGP: Total Graphics Power
- TPC: Tensor Processing Core
- TPU: Tensor Processing Unit
- WSE: Wafer Scale Engine - Cerebras' whole-wafer accelerator
- XCD: Accelerator Complex Die - AMD's per-die cache reporting scope
- XLA: Accelerated Linear Algebra - the compiler behind PyTorch/JAX on TPUs and Trainium

Additional glossary @ Modal

The most important thing to understand

I will make the following statement multiple times in this book: it's not enough to buy/rent the most expensive accelerators and expect a high return on investment (ROI).

The two metrics for a high ROI for ML training are:

1. the speed at which the training will finish, because if the training takes 2-3x longer than planned, your model could become irrelevant before it was released - time is everything in the current super-competitive ML market.
2. the total $$ spent to train the model, because if the training takes 2-3x longer than planned, you will end

up spending 2-3x times more.

If the rest of the purchased/rented hardware isn't chosen carefully to match the required workload, chances are very high that the accelerators will idle a lot and both time and $$ will be lost. The most critical component is network, then storage, and the least critical ones are CPU and CPU memory (at least for a typical training workload where any CPU limitations are compensated with multiple `DataLoader` workers).

If the compute is rented one usually doesn't have the freedom to choose - the hardware is either set in stone or some components might be replaceable but with not too many choices. Thus there are times when the chosen cloud provider doesn't provide a sufficiently well matched hardware, in which case it's best to seek out a different provider.

If you purchase your servers then I recommend to perform a very indepth due diligence before buying.

Besides hardware, you, of course, need software that can efficiently deploy the hardware.

We will discuss both the hardware and the software aspects in various chapters of this book. You may want to start here and here.

What Accelerator characteristics do we care for

Let's use the NVIDIA A100 spec as a reference point in the following sections.

	A100 80GB PCIe	A100 80GB SXM
FP64	9.7 TFLOPS	
FP64 Tensor Core	19.5 TFLOPS	
FP32	19.5 TFLOPS	
Tensor Float 32 (TF32)	156 TFLOPS | 312 TFLOPS*	
BFLOAT16 Tensor Core	312 TFLOPS | 624 TFLOPS*	
FP16 Tensor Core	312 TFLOPS | 624 TFLOPS*	
INT8 Tensor Core	624 TOPS | 1248 TOPS*	
GPU Memory	80GB HBM2e	80GB HBM2e
GPU Memory Bandwidth	1,935 GB/s	2,039 GB/s
Max Thermal Design Power (TDP)	300W	400W ***
Multi-Instance GPU	Up to 7 MIGs @ 10GB	Up to 7 MIGs @ 10GB
Form Factor	PCIe Dual-slot air-cooled or single-slot liquid-cooled	SXM
Interconnect	NVIDIA® NVLink® Bridge for 2 GPUs: 600 GB/s ** PCIe Gen4: 64 GB/s	NVLink: 600 GB/s PCIe Gen4: 64 GB/s
Server Options	Partner and NVIDIA-Certified Systems™ with 1-8 GPUs	NVIDIA HGX™ A100-Partner and NVIDIA-Certified Systems with 4,8, or 16 GPUs NVIDIA DGX™ A100 with 8 GPUs

source

TFLOPS

Most of the work that ML training and inference do is matrix multiplication. If you remember your algebra matrix multiplication is made of many multiplications followed by summation. Each of these computations can be counted and define how many of these operations can be performed by the chip in a single seconds.

This is one of the key characteristics that the accelerators are judged by. The term TFLOPS defines how many trillions of FloatingPointOperations the chip can perform in a second. The more the better. There is a different definition for different data types. For example, here are a few entries from the theoretical peak TFLOPS from A100 spec:

Data type \ TFLOPS	w/o Sparsity	w/ Sparsity
FP32	19.5	n/a
Tensor Float 32 (TF32)	156	312
BFLOAT16 Tensor Core	312	624
FP16 Tensor Core	312	624
FP8 Tensor Core	624	1248
INT8 Tensor Core	624	1248

Notes:

- INT8 is measured in TeraOperations as it's not a floating operation.
- the term FLOPS could mean either the total number of FloatingPointOperations, e.g. when counting how many FLOPS a single Transformer iteration takes, and it could also mean FloatingPointOperations per second - so watch out for the context. When you read an accelerator spec it's almost always a per second definition. When model architectures are discussed it's usually just the total number of FloatingPointOperations.

So you can see that int8 is 2x faster than bf16 which in turn is 2x faster than tf32.

Moreover, the TFLOPS depend on the matrices size as can be seen from this table:

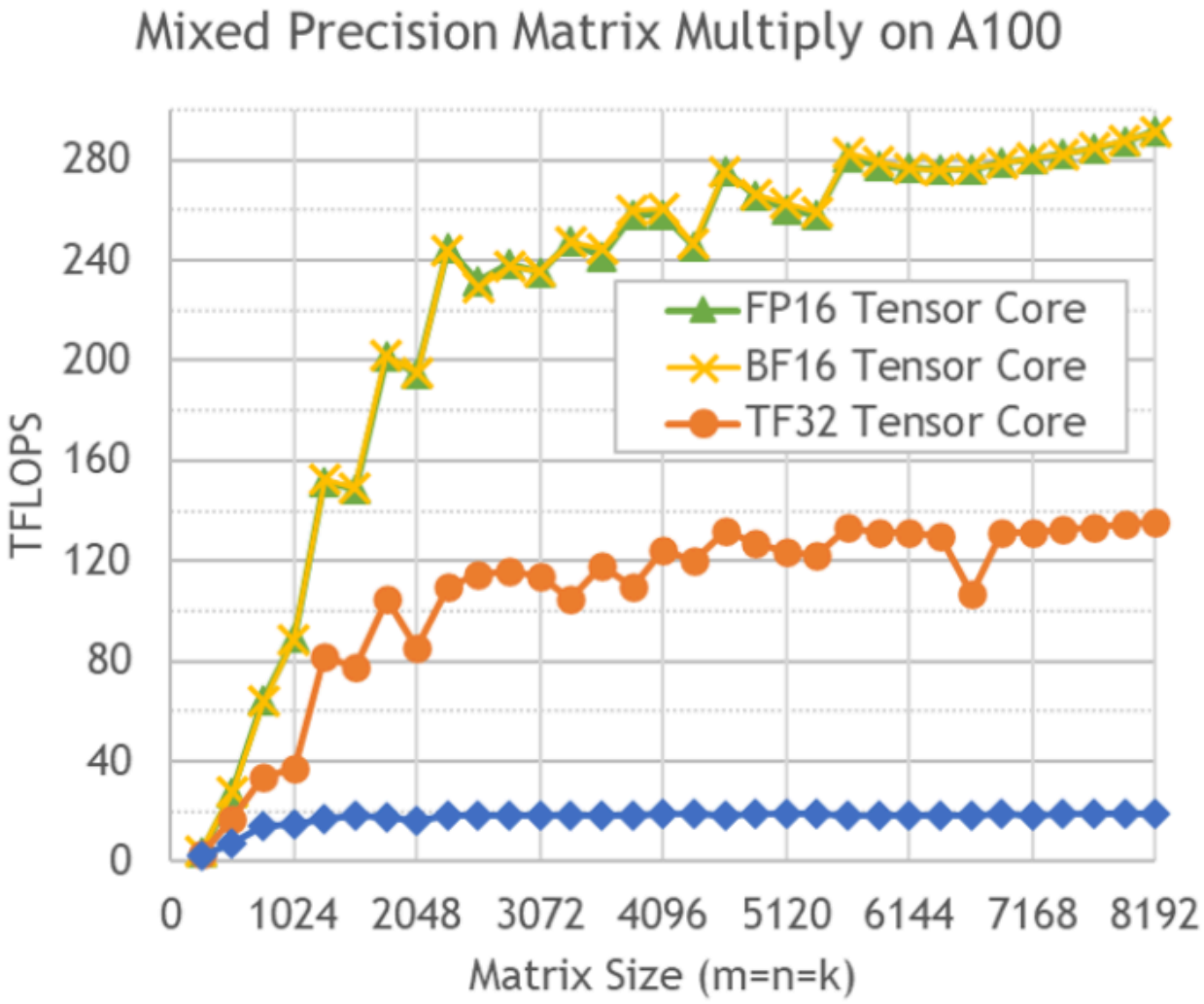


Figure 6. Mixed-precision matrix multiply on A100 with cuBLAS.

source

As you can see the difference in performance is non-linear due to the tile and wave quantization effects. Note the blue line in the graph corresponds to FP32 Tensor Core.

How To Calculate Theoretical TFLOPS

Theoretical peak FLOPS is what gets published on the accelerator's spec. And it's calculated as:

```
Theoretical FLOPS = compute_unit_clock_speed * FLOPs_per_clock_cycle_per_compute_unit * num_compute_units
```

where:

- `compute_unit_clock_speed` - how many times the compute unit clock ticks per second in Hz
- `flops_per_clock_cycle_per_compute_unit` - the number of floating point operations the compute unit can execute per clock cycle.
- `num_compute_units` - how many units there is in the device

FLOPs per clock cycle per compute unit is usually not published, but what one often finds is the FMAs per clock cycle per compute unit specs. FMA is Fused Multiply Add. And since 1 FMA is made of 2 FLOPs, we can expand the above formula to:

`Theoretical FLOPS = compute_unit_clock_speed * FMAs_per_clock_cycle_per_compute_unit * 2 * num_compute_units`

Let's validate that this formula checks out. Let's compute some BF16 (half precision) TFLOPS and compare to the published specs.

First, let's extract the necessary accelerator specs from wiki.

The tricky part was to find the FMAs ops per Tensor Core per clock cycle for BF16 (half precision). I found them here. Most are coming from the A100 whitepaper (search the pdf for "FMA" and then choose the ones listed for the target precision you're after). The H100 whitepaper omitted a lot of specific FMA numbers, but included the multipliers wrt FMAs listed in the A100 whitepaper).

The clock speeds used in the following calculations and what each vendor's clock actually means, are collected in Clock speed.

For NVIDIA @ BF16:

For NVIDIA BF16 operations are performed by Tensor cores.

Accelerator	Clock	FMAs ops per Tensor Core per clock cycle	Tensor Cores	Spec TFLOPS	Notes
H100 SXM	1830MHz	512	528	989	1
A100 SXM	1410MHz	256	432	312	1

1. `Tensor Cores` is derived: 4 Tensor Cores per SM, times 132 SMs for H100 SXM and 108 SMs for A100 SXM.

Now let's do the math, by inserting the numbers from the table above into the last FMA-based formula:

- `1830*10**6 * 512 * 2 * 528 / 10**12 = 989.430` TFLOPS
- `1410*10**6 * 256 * 2 * 432 / 10**12 = 311.87` TFLOPS

Both calculated numbers match the published specs - 989 for H100 SXM and 312 for A100 SXM.

Note that this peak is an aggregate over the whole accelerator - `528` Tensor Cores is `4 x 132` SMs - so it is not what a single matmul achieves unless that matmul is large enough to occupy every SM. For why a part with more SMs does not automatically deliver more TFLOPS, see Do more SMs give more TFLOPS?.

The H100 SXM clock above is 1830MHz, which is now NVIDIA's official figure. Earlier 1980MHz was the widely published number, and it doesn't work out: `1980*10**6 * 512 * 2 * 528 / 10**12 = 1070.530` TFLOPS, some 80 points above the spec. Before 1830MHz became official you could already recover it by inverting the same formula against the spec TFLOPS: `989 / (512 * 2 * 528 / 10**12) / 10**6 = 1829.20`. Which is a useful trick in general - if a vendor publishes the TFLOPS but not the clock, or publishes a clock that doesn't reproduce their own TFLOPS, the formula tells you which number they actually used.

For AMD @ BF16:

Accelerator	Peak Engine Clock	FMAs ops per Matrix Core per clock cycle	Matrix Cores	Spec TFLOPS	Notes
MI355X	2400MHz	512	1024	2500	1
MI300X	2100MHz	256	1216	1307	
MI250X	1700MHz	128	880	383	2

1. AMD publishes `2.5 PFLOPS` for MI355X BF16 matrix, rounded to two significant digits - the formula gives

2516.6.
2. AMD publishes the Matrix Core count for MI355X (1024) and MI300X (1216), but not for MI250X, so its 880 is derived: 4 Matrix Cores per CU, times 220 CUs.

Let's calculate ourselves as before:

- `2400*10**6 * 512 * 2 * 1024 / 10**12 = 2516.6` TFLOPS - matches AMD's published `2.5 PFLOPS` within their own rounding
- `2100*10**6 * 256 * 2 * 1216 / 10**12 = 1307.4` TFLOPS - matches the published spec, even though most of the time you will see the rounded down `1300` TFLOPS in the literature.
- `1700*10**6 * 128 * 2 * 880 / 10**12 = 383.0` TFLOPS - matches the published spec exactly

Note the FMAs per Matrix Core doubling every CDNA generation - 128 on MI250X (CDNA2), 256 on MI300X (CDNA3), 512 on MI355X (CDNA4). Since the Matrix Core count barely moved (880 to 1216 to 1024), that per-core doubling is where almost all of the generational BF16 gain came from, not from more cores.

For Intel @ BF16:

Intel Gaudi uses MMEs to do BF16 `matmul`

Accelerator	MME Clock	FMAs ops per MME per clock cycle	MMEs	Spec TFLOPS
Gaudi 2	1650MHz	256*256	2	432
Gaudi 3	1600MHz	256*256	8	1677

Let's calculate ourselves as before:

- Gaudi 2: `1650*10**6 * 256*256 * 2 * 2 / 10**12 = 432.5` TFLOPS - matches the published spec
- Gaudi 3: `1600*10**6 * 256*256 * 2 * 8 / 10**12 = 1677` TFLOPS - note that this doesn't match the published spec in the whitepaper (1835TFLOPS), because in order to have 1835TFLOPS the clock has to be 1750MHz. i.e. the current incarnation of Gaudi3 is running at 1600MHz.

It should become obvious now that if your accelerator runs at a lower boost clock than the spec (e.g. overheating that leads to accelerator throttling) the expected TFLOPS will be lower than advertised.

To check the actual clock speed when your accelerator is under load see the clock speed section.

TFLOPS comparison table

Let's look at the supported dtypes and the corresponding theoretical peak TFLOPS specs across the high end accelerators (w/o sparsity). Both tables are sorted by the bf16 column, and they share one set of notes, listed after them.

Generally available:

Accelerator \ TFLOPS	fp32	tf32	fp16	bf16	fp8	int8	fp6	fp4	nvfp4	Notes
NVIDIA GB300 SXM	80.0	1250.0	2500	2500	5000	5000	5000	15000	?	24
NVIDIA GB200 SXM	80.0	1250.0	2500	2500	5000	5000	5000	10000	?	23
AMD MI355X	157.3	?	2500	2500	5000	5000	10100	10100	X	22
Google TPU v7x	?	?	2307	2307	4614	?	?	?	?	25
NVIDIA B300 SXM	80.0	1125.0	2250	2250	4500	4500	4500	12600	15000	21
NVIDIA B200 SXM	80.0	1125.0	2250	2250	4500	4500	4500	9000	10000	20

Accelerator \ TFLOPS	fp32	tf32	fp16	bf16	fp8	int8	fp6	fp4	nvfp4	Notes
Intel Gaudi3	229.0	459.0	459	1677	1677	V	X	X	X	8,19
AMD MI325X	163.4	653.7	1300	1300	2610	2600	X	X	X	18
AMD MI300X	163.4	653.7	1300	1300	2610	2600	X	X	X	17
NVIDIA H200 SXM	67.0	494.5	989	989	1979	1979	X	X	X	16
NVIDIA H100 SXM	67.0	494.5	989	989	1979	1979	X	X	X	12,15
NVIDIA GH200 SXM	67.0	494.5	989	989	1979	1979	X	X	X	14
Google TPU v6e	?	?	918	918	918	1836	X	X	X	13
NVIDIA H100 PCIe	51.0	378.0	756	756	1513	1513	X	X	X	12
AWS Trainium2 / Ultra	181.0	667.0	667	667	1299	X	X	X	X	11
Google TPU v5p	X	X	X	459	X	918	X	X	X	10
Intel Gaudi2	V	V	V	432	865	V	X	X	X	8,9
AMD MI250X	47.9	X	383	383	X	383	X	X	X	7
NVIDIA L40S	91.6	183.0	362	362	733	733	X	X	X	6
AMD MI250	45.3	X	362	362	X	362	X	X	X	5
NVIDIA A100 SXM	19.5	156.0	312	312	X	624	X	X	X	4
NVIDIA A100 PCIe	19.5	156.0	312	312	X	624	X	X	X	3,4
Google TPU v4	X	X	X	275	X	275	X	X	X	2
Google TPU v5e	X	X	X	197	X	394	X	X	X	1

Announced, availability not confirmed:

Accelerator \ TFLOPS	fp32	tf32	fp16	bf16	fp8	int8	fp6	fp4	nvfp4	Notes
AMD MI455X	315.0	?	5000	5000	10000	5000	10000	20000	X	27
NVIDIA Rubin SXM	130.0	2000.0	4000	4000	17500	2500	17500	35000	35000	26
Huawei Ascend 950DT	X	X	486	486	919	X	X	1783	X	28

Notes and sources - the `Notes` column of both tables points here. Numbers run from the oldest hardware upward, so anything new gets the next number and lands at the end of this list:

1. Google Cloud TPU v5e documentation
2. Google Cloud TPU v4 documentation
3. Oddly NVIDIA A100 PCIe and SXM revisions spec are reported to have the same TFLOPS, which is odd considering the SXM version uses 30% more power and uses a 5% faster HBM.
4. NVIDIA A100 specifications - covers both the SXM and PCIe rows
5. AMD Instinct MI250 specifications

6. NVIDIA L40S specifications
7. AMD Instinct MI250X specifications
8. Intel Gaudi2 and 3 only have partial TFLOPS specs published, but it does support FP32, TF32, BF16, FP16 & FP8, INT8 and INT16. These numbers are for MME (Matrix) compute.
9. Intel Gaudi 2 AI accelerator white paper
10. Google Cloud TPU v5p documentation
11. Trainium2 also supports FP8/FP16/BF16/TF32 @ 2563TFLOPS w/ 4:1 sparsity. See AWS Trainium2 architecture documentation.
12. NVIDIA H100 specifications - covers both the SXM and PCIe rows
13. Google Cloud TPU v6e documentation
14. GH200 - same note as GB200 - this is 2 GPUs in one package, so the table includes specs per chip w/o sparsity. See NVIDIA GH200 Grace Hopper Superchip specifications.
15. I didn't include `NVIDIA H100 dual NVL` as it's, well, 2x GPUs - so it won't be fair - it's the same FLOPS as H100 but 2x everything, plus at has a bit more memory (94GiB per chip, as compared to 80GiB H100) and the memory is a bit faster.
16. H200 is the same as H100 but has 141GiB vs 80GiB of HBM memory, and its memory is faster, HBMe@4.8TBps vs HBM@3.35TBps - so basically H200 solves the memory-bandwidth and memory-capacity bottlenecks of H100. See NVIDIA H200 specifications.
17. AMD Instinct MI300X specifications
18. MI325X is the same compute as MI300X, but has more memory and more power (more efficient compute). See AMD Instinct MI325X specifications.
19. Gaudi3 as of 2026-08 is running at 1600MHz (MME) and not the planned 1750MHz, therefore its BF16 TFLOPS are 1677 and not 1835 as per whitepaper spec. Same goes for fp8 which runs at the same TFLOPS as BF16.
20. NVIDIA DGX B200 datasheet
21. NVIDIA DGX B300 datasheet
22. AMD Instinct MI355X specifications - these are AMD's dense figures. AMD also publishes `with Structured Sparsity` variants at exactly 2x - 10.1PFLOPS for OCP-FP8 and 5PFLOPS for FP16 matrix - so a 10.1PFLOPS fp8 number quoted elsewhere is the sparse one, not this table's. The `fp6` and `fp4` entries carry no sparsity qualifier on AMD's page and are dense. MI350X is the same silicon at 2200MHz and 1000W, with everything scaled by the clock ratio (144.2 vs 157.3 fp32); it is left out because MI355X is the part you can actually rent.
23. Since GB200 is 2x B200 chips the table includes TFLOPS per chip for a fair comparison - you'd 2x it for the real GB200 - it also seems to run the B200 chips a bit faster so higher specs than standalone B200. This also means that instead of your typical 8-GPU node, with GB200 you will get a 4-GPU node instead (but it'd be the equivalent of 8x B200 w/ an additional ~10% faster compute). See NVIDIA GB200 NVL72 specifications.
24. GB200 NVL72 and GB300 NVL72 seem to be the same but faster fp4 and more memory for the latter. See NVIDIA GB300 NVL72 specifications.
25. Google Cloud TPU v7x documentation - Google calls it "the latest TPU available on Google Cloud" and documents using it through GKE or Compute Engine, so it is treated as available like every other TPU here, all of which are rent-only and capacity-gated. Only fp16, bf16 and fp8 are published; the rest of the row is `?` because Google has not stated those numbers.
26. NVIDIA Vera Rubin NVL72 specifications - pre-release, so these are announced figures rather than a shipped part's spec sheet
27. AMD's MI455X figures come from its product page, read on 2026-08-06. `fp32` 315, `fp16` 5000, `bf16` 5000 and `int8` 5000 are AMD's own dense numbers. `fp8`, `fp6` and `fp4` are halved from the 20100, 20100 and 40300 that AMD publishes, because those three are almost certainly with-sparsity figures that AMD did not declare as such: it labels the sparsity variants of FP16, BF16 and INT8 explicitly on the same page, all at 10100, and taking the three unlabelled numbers at face value would make fp8 4x bf16 where every other accelerator here is 2x - MI355X runs 2500, 5000, 10100. Halving restores that doubling exactly. AMD publishes no TF32 figure for this part, hence `?`. All of it remains AMD Performance Labs projections carrying "Results subject to change when products are released in market".
28. Ascend 950DT figures are from the Atlas 950 SuperPoD product-spec table on Ascend Community (read

2026-08-10): `114.1 / 58.8 / 31.1` PFLOPS @ `mxFP4` / `mxFP8/HiF8` / `FP16/BF16` for **64×** Ascend 950DT, divided by 64 → **1783 / 919 / 486**. Do not mix with the 1024-card SuperPoD totals on the Chinese page (those give ~1953 / 977 mxFP4/mxFP8 and no bf16). The `fp8` column is Huawei's single published `mxFP8/HiF8` rate; HiF8 is tapered-precision, not OCP E4M3/E5M2, so the chapter's `e/m` dtype decoder does not apply to it. Unpublished columns are `x`. General notes:

- int8 is measured in TeraOperations as it's not a floating operation.
- if you find numbers that are double of the above - it usually means with sparsity (which at the moment almost nobody can benefit from as our matrices are dense).
- when looking at specs be very careful at which numbers you're reading - many vendors often publish TFLOPS with sparsity, as they are ~2x bigger, but if they even indicate this they often do it in small print. I had to ask NVIDIA to add a note to their H100 spec that those numbers were w/ sparsity as they originally didn't mention this important technical fact. And 99% of the time as of 2026-08 you will be not using sparsity and thus the actual theoretical TFLOPS that you care for most of the time are w/o sparsity (i.e. the table above).
- also beware that if accelerator A publishes a higher TFLOPS than accelerator B, it doesn't mean A is faster. These are theoretical numbers which not only can never be achieved in practice - the actual TFLOPS efficiency Hardware FLOPS Utilization (HFU) can vary a lot from vendor to vendor or even for the same vendor's different accelerator architectures.

Maximum Achievable and Sustainable FLOPS

The problem with the advertised theoretical peak FLOPS is that they are **very** theoretical and can't be achieved in practice even if all the perfect conditions have been provided. Each accelerator has its own realistic FLOPS which is not advertised and there are anecdotal community reports that do their best to find the actual best value, but I'm yet to find any official reports.

If you find solid reports (papers?) showing the actual TFLOPS one can expect from one or more of the high end accelerators discussed in this chapter please kindly submit a PR with this information. The key is to have a reference to a source that the reader can validate the proposed information with.

To provide a numerical sense to what I'm talking about let's take an A100 with its 312TFLOPS bf16 peak performance in the specs of this card. Until the invention of FlashAttention it was known that 150TFLOPS was close to the highest one could get for fp16/bf16 mixed precision training regime. And with FlashAttention it's around 180+TFLOPS. This is, of course, measured for training LLMs where the network and IO are involved which create additional overheads. So here the maximum achievable peak performance probably lays somewhere between 200 and 300TFLOPS.

You could measure the actual achievable peak TFLOPS by doing a perfectly aligned max-size matrices `matmul` measured on a single accelerator. You can use Maximum Achievable and Sustainable Matmul FLOPS finder to reproduce the results. But, of course, this will only tell you how well your given accelerator and its software stack do `matmul` - depending on the workload this might be all you need to know, or not.

MAMF stands for Maximum Achievable Matmul FLOPS, which is a term coined by yours truly, and MSMF for Maximum Sustainable Matmul FLOPS - what the accelerator holds once it is saturated at its power limit. Both are very practical for those who do performance optimization work.

Maximum Achievable and Sustainable Matmul FLOPS comparison table

The following measurements are for `matmul` with BF16 and FP8 inputs (no sparsity) TFLOPS (see Maximum Achievable and Sustainable FLOPS for what MAMF and MSMF stand for). **MAMF** is the short boost-clock burst; **MSMF** is what it holds once the chip is saturated at its power limit — use MSMF for training throughput. `–` means MSMF is not measured yet. Sorted by MAMF %. Reproduce with the MAMF finder.

BF16:

Accelerator	MAMF	MSMF	Theory	MAMF %	MSMF %	Sib lings	Shape MxNxK (MAMF)	torch version	Notes
Intel Gaudi 2	419	—	432	97.0%	—	no	14336x15360x2048	2.6.0+hpu_1.21.2-76	1
NVIDIA A100 PCIe	284	255	312	91.0%	81.7%	no	18432x1536x20480	2.12.1+cu130	2
NVIDIA A100 SXM	271	—	312	86.9%	—	no	1024x10240x5120	2.6.0+cu126	
NVIDIA H200 SXM	834	755	989	84.3%	76.3%	no	3072x2816x16384	2.14.0+cu130	3
NVIDIA B300 SXM	1892	1519	2250	84.1%	67.5%	no	10752x14336x3072	2.14.0+cu130	4
NVIDIA GH200 SXM	829	—	989	83.8%	—	no	1024x15360x4096	2.6.0+cu126	5
NVIDIA H100 SXM	795	—	989	80.4%	—	no	2048x2048x13312	2.7.0+cu126	
NVIDIA B200 SXM	1703	1429	2250	75.7%	63.5%	yes	4864x8960x2048	2.14.0+cu130	6
Intel Gaudi 3	1243	—	1677	74.1%	—	no	16384x4096x768	2.6.0+hpu_1.21.4-3	1
NVIDIA GB200 SXM	1822	—	2500	72.9%	—	no	4096x9728x2048	2.10.0+cu130	
AMD MI355X	1565	—	2500	62.6%	—	no	12288x8192x8192	2.8.0+rocm7.0.2	7
AMD MI325X	785	—	1300	60.4%	—	no	13312x10240x8192	2.6.0+6.2.4	8
AMD MI300X	676	659	1300	52.0%	50.7%	no	19456x3072x8192	2.12.0+rocm10.0.0	9

FP8 (`float8_e4m3fn`):

Accelerator	MAMF	MSMF	Theory	MAMF %	MSMF %	Sib lings	Shape MxNxK (MAMF)	torch version	Notes
Intel Gaudi 2	827	—	865	95.6%	—	no	6144x11264x5120	2.6.0+hpu_1.21.2-76	1
NVIDIA B300 SXM	3608	2969	4500	80.2%	66.0%	no	6144x18432x3072	2.14.0+cu130	10
NVIDIA GH200 SXM	1535	—	1979	77.6%	—	no	1024x14336x14336	2.6.0+cu126	5
Intel Gaudi 3	1290	—	1677	76.9%	—	no	16640x1536x3072	2.6.0+hpu_1.21.4-3	1
NVIDIA H200 SXM	1503	1290	1979	75.9%	65.2%	no	3840x2816x20480	2.14.0+cu130	11

Accelerator	MAMF	MSMF	Theory	MAMF %	MSMF %	Sib lings	Shape MxNxK (MAMF)	torch version	Notes
NVIDIA B200 SXM	3331	2829	4500	74.0%	62.9%	yes	6144x11520x2048	2.14.0+cu130	12
NVIDIA GB200 SXM	3616	—	5000	72.3%	—	no	19456x5120x1536	2.10.0+cu130	
NVIDIA H100 SXM	1403	—	1979	70.9%	—	no	1024x9216x14336	2.7.0+cu126	
AMD MI300X	1201	1187	2600	46.2%	45.7%	no	4096x19456x16384	2.12.0+rocm10.0.0	13

Caveat emptor: these numbers come from `mamf-finder.py --search auto` (or grid confirm of the same / equivalent shapes) using the software stack available at measurement time. Re-run on your setup for numbers that are true to your box — they are a rough estimate, not absolute. As software improves they climb toward the theoretical spec, so ideally re-measure every 6 months or so.

Notes — the `Notes` column of both tables points here:

1. Intel Gaudi 2 and Intel Gaudi 3 (both tables): `PT_HPU_LAZY_MODE=1`
2. NVIDIA A100 PCIe, BF16: 300 W; MAMF @ 1395 MHz; MSMF 255 @ 1152x3072x11264, ~298 W @ 1260 MHz
3. NVIDIA H200 SXM, BF16: MAMF ~169 W @ 1980 MHz; MSMF 755 @ 1536x2816x16384, ~688 W @ 1695 MHz
4. NVIDIA B300 SXM, BF16: B300 SXM6 AC 1100 W; MAMF ~286 W @ 2032 MHz; MSMF 1519 @ 8192x18432x1024, ~1066 W @ 1455 MHz
5. NVIDIA GH200 SXM (both tables): 900W 141GiB HBM3e version
6. NVIDIA B200 SXM, BF16: MAMF ~295 W @ 1965 MHz; MSMF 1429 @ 2560x16896x4096, ~978 W @ 1455 MHz
7. AMD MI355X, BF16: `PYTORCH_TUNABLEOP_ENABLED=0`
8. AMD MI325X, BF16: `PYTORCH_TUNABLEOP_ENABLED=1`, 1000W
9. AMD MI300X, BF16: `PYTORCH_TUNABLEOP_ENABLED=1`, 750 W; MAMF ~210 W @ 2067 MHz; MSMF 659 @ 12288x9728x8192, 750 W @ 1290 MHz
10. NVIDIA B300 SXM, FP8: MAMF ~262 W @ 2032 MHz; MSMF 2969 @ 6144x18432x3072, ~1057 W @ 1425 MHz
11. NVIDIA H200 SXM, FP8: MAMF ~142 W @ 1980 MHz; MSMF 1290 @ 3840x2816x20480, ~690 W @ 1440 MHz
12. NVIDIA B200 SXM, FP8: MAMF ~290 W @ 1965 MHz; MSMF 2829 @ 6144x12288x3072, ~965 W @ 1485 MHz
13. AMD MI300X, FP8: `float8_e4m3fnuz`, `PYTORCH_TUNABLEOP_ENABLED=1`, 750 W; MAMF ~218 W @ 2058 MHz; MSMF 1187 @ 6912x19200x16384, 750 W @ 1189 MHz

General notes:

- For the full set of theoretical ones see Theoretical accelerator TFLOPS
- `MAMF %` / `MSMF %` are `MAMF/Theory*100` and `MSMF/Theory*100`
- While `mean` is probably what most users are interested in, the script reports `max`, `median` and `mean` - should you want the other numbers.
- The `Shape` column is the MAMF shape; the MSMF shape, its power and its clock are in the numbered notes. There are usually many shapes with near-identical performance — these are listed for reproducibility.
- `Sib` = `yes`: the other GPUs of the node ran a continuous matmul during the measurement, and MSMF is the slowest GPU. `Sib` = `no`: measured alone, so MSMF is a single-GPU upper bound. How `mamf-finder.py` measures either is explained here.
- If you get a much lower performance than the numbers in this table, check that the target hardware has an adequate cooling, if the accelerator is overheated it'd usually throttle its performance down. And, of course, the assumption here is that the power supply matches the spec. The latter is rarely a problem in data centers, but bad cooling is not unheard of.
- Which software you use can make a huge difference - e.g., with MI300X I clocked 450TFLOPS using

ROCm-6.1, but as you can see there was a dramatic improvement in ROCm-6.2 where it jumped a whooping additional 300TFLOPS up. BLAS library type/version may have a big impact as well.

- Then there are various system optimizations - e.g. in the case of MI300X disabling numa_balancing in the kernel settings is a must.
- Rows are not always exactly comparable with each other. Each is the best shape found for *that* accelerator, and often on a different `torch`/CUDA or ROCm version, so any gap between two rows mixes hardware, shape and software. But it's a good enough of indication to compare with the theoretical spec. As software evolves remeasuring is needed and likely to give better results, but yours truly doesn't have access to all the gpus, especially the older ones, thus contributions are very welcome.
- AMD MI250X has 2 GCDs - so the theoretical TFLOPS needs to be halved, as a single matmul uses only 1 of them and 383TFLOPS is reported for 2 GCDs.

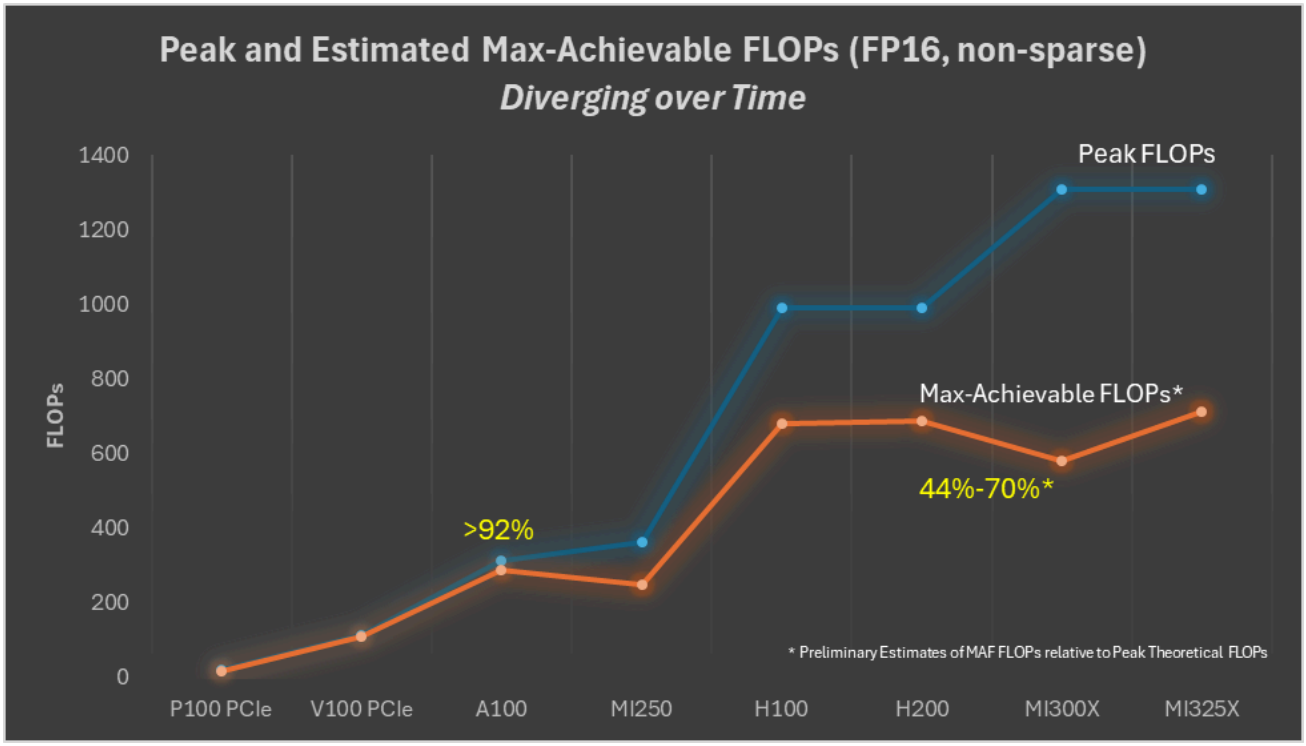
Why MSMF is well below Theory (and why that is not a broken GPU): the advertised peak assumes the chip's *boost* clock (e.g. B200 ~1965 MHz → 2250 bf16 TFLOPS). A dense matmul that actually fills the SMs draws the full TDP (B200 1000 W / B300 1100 W / H200 700 W — those are the *spec* limits, not a sub-spec "cap"), so the SM clock settles well below boost. Rough ceiling: `Theory × sustained_clk / boost_clk` ≈ `2250 × 1425/1965 ≈ 1632` on B200 — in the ballpark of the measured ~1429 MSMF (the rest is real-kernel overhead). A sparse layout can hold a *higher* saturated clock while still pinned at TDP (H200 MSMF at 1695 MHz, B300 MSMF at 1455 MHz); that number is still sustainable.

Why MAMF sits above MSMF: MAMF is a short unsaturated burst. The two headlines often land on *different* shapes — a fat GEMM can be the better boost burst, a skinnier one the better sustained rate (or the reverse). Concretely (B300): `10752×14336×3072` bursts at **2032 MHz / 286 W → 1892 TFLOPS** (MAMF) but holds only 1487 TFLOPS once saturated; `8192×18432×1024` bursts lower at 1809 yet holds **1455 MHz / 1066 W → 1519 TFLOPS** (MSMF). `mamf-finder.py` times a queued burst after an idle and reports the peak iteration only if the clock sampled across that iteration reached boost, so a throttled/base-clock reading can't masquerade as MAMF. After both headlines it prints each winning shape in both regimes.

Also it's important to understand that knowing the Maximum Achievable Matmul TFLOPS at some particular shape like `4352x3840x13568` doesn't mean you can expect to get the same performance in your real application because chances are low that you will ever hit that exact shape. Instead, to know your system well, you'd run the MAMF Finder with the actual shapes your model is using during its training. This really is the main intention of this tool. You will have a good sense of when you can stop optimizing by comparing the TFLOPS reported by your training to Maximum Achievable MatMul TFLOPS you measured on your specific accelerator cluster.

And to conclude this section I'd like to repeat again that **the intention here is not to point fingers at which accelerator is less efficient than another, but to give a sense of what's what and how to navigate those theoretical specs and to help you understand when you need to continue optimizing your system and when to stop. So begin with these notes and numbers as a starting point, then measure your own use case and use that latter measurement to gain the best outcome.**

update: this new metric is starting to catch on. AMD published this graph and explanations of why the efficiency of accelerators is going down as they get faster:



source

Not all accelerators are created equal

While measuring how well an accelerator performs, you need to be aware that while it gives you the ballpark performance numbers, other accelerators are likely to perform slightly differently. I have seen 5% and higher differences on an 8-gpu node.

This partially has to do with manufacturing processes, how well each accelerator is installed and much more about how equally each accelerator is cooled. For example, when air cooling is used it's very likely that the accelerators closer to the source of cooling will perform better than those further away, especially since now the hot air dissipated from one row gets blown into the next row of accelerators. Things should be better with liquid cooling.

Silicon lottery

Even two chips coming off the same wafer are not identical. Tiny variations during manufacturing mean each die ends up with slightly different transistor characteristics. Some leak more current, some need higher voltage to switch reliably at a given frequency, others can run cooler or hotter at the same workload. This is known as the **silicon lottery**: you can buy two GPUs with the same model number and SKU and get measurably different performance and power behavior from them.

The way this is characterized is the per-chip **V-F-T curve** (Voltage-Frequency-Temperature). Every die has its own surface describing the minimum voltage required to run stably at a given frequency and temperature. A "good" die can hit higher frequencies at lower voltage (less heat, less power, more headroom before throttling); a "weaker" die needs more voltage for the same clock, which produces more heat, which then forces the clock down sooner under load. So the curves are not parallel. Two dies that are equal at idle can diverge significantly once they're hot and under sustained compute.

In data-center accelerators the boost-clock and TDP spec is set conservatively so that every binned chip can meet it, but the *actual sustained performance* still falls anywhere on a distribution. Combined with uneven cooling (see above), this explains why on the same 8-GPU node you can see 5%+ MAMF spread between GPUs even when all of them pass the same factory binning.

Therefore, you want to measure the performance of all accelerators on the node and do it at the same time. For example,

on NVIDIA nodes, if each benchmark measures a single accelerator, you could do:

```
CUDA_VISIBLE_DEVICES=0 ./some-benchmark
CUDA_VISIBLE_DEVICES=2 ./some-benchmark
...
CUDA_VISIBLE_DEVICES=7 ./some-benchmark
```

Now here what you want is the slowest performance as when used in an ensemble that slowest accelerator (struggler) will set the speed for all other accelerators.

If you do multi-node training then, of course, you'd want to measure them all.

So if you decide to calculate your achievable MFU (rather than theoretical one) you'd want to measure the achievable FLOPS across all participating accelerators and pick the value of the slowest accelerator. (If it really is an outlier you might want to consider replacing it as well).

Accelerator memory size and speed

The accelerators use High Bandwidth Memory (HBM) which is a 3D version of SDRAM memory. For example, the 80GiB A100-SXM comes with HBM2e at 2.0TBps, and the 80GiB H100-SXM comes with HBM3 at 3.35TBps (see the full table per accelerator below).

Here are the specs:

Type	Max data rate speed per pin (Gbps)	Stack Height	Bits per Channel	Number of dies per stack	Die capacity per stack (GiBs)	Max capacity per stack (GiBs)	Max data rate per stack (GBps)
HBM1	1.0	8	128	4	1	4	128
HBM2	2.4	8	128	8	1	8	307
HBM2e	3.6	8	128	12	2	24	461
HBM3	6.4	16	64	12	2	24	819
HBM3e	9.6	16	64	16	3	48	1229
HBM4	8.0	32	64	16	4	64	2048

Notes:

- While I was researching this table I found a wide variation of the above numbers. I think it's because either there were different implementations or the specs changed several times and different publications caught different specs. The table above comes from wikipedia.
- Since HBM is a stack of multiple DRAM chips, the *Stack Height* specifies how many chips are per device.

Beware that sometimes memory specs may not be very clear about what GB means. Sometimes it's GiB (`2**30` bytes), but written as GB. At other times it's actually GB (`10**9` bytes). Bandwidth (`GBps`, `TBps`) almost always means decimal units (`1GBps = 10**9` bytes per second). To convert GiB to GB: `x * 2**30 / 10**9`. To convert from GB to GiB: `x * 10**9 / 2**30`. Most often, memory size will be in GiB (with the `i` omitted), while bandwidth will use decimal GBps or TBps.

To make things even more confusing, the advertised capacity isn't always what you actually get to use. One possible explanation for NVIDIA B200's 192GB and 180GB figures is that it physically has 192GiB of HBM3e - 8 stacks of 24GiB each

(these are GiB and not decimal GB: DRAM die density is inherently binary, so a stack of 8x 24Gbit dies is `8*24*2**30/8 = 24GiB`, and 8 stacks give 192GiB - vendors just print it as "192GB"). Under this interpretation, only about 180GiB is usable in HGX/DGX systems and the rest is reserved, e.g. for ECC; `192*15/16 = 180` exactly, suggesting that 1/16 is held back. So the 192-versus-180 difference would be physical versus usable capacity, not GB versus GiB, and this is why the table below lists NVIDIA's 180GB hardware/platform figure. However, the 192GiB physical capacity, eight-stack layout, binary-unit interpretation, and 1/16 reservation are derived from arithmetic: NVIDIA has not officially documented this relationship, and the actual hardware layout or memory accounting could differ.

Typically, the more on-device memory an accelerator has, the better its performance. At any given time usually most of the model weights aren't being used as they wait for their turn to be processed and thus large memory allows more of the model to be on the accelerator memory and immediately available for access and update. When there is not enough memory, sometimes the model has to be split across multiple accelerators, or offloaded to CPU and/or disk.

Here are the memory specs for the recent high end accelerators (some aren't GA yet), sorted by memory size, then bandwidth:

Generally available:

Accelerator	Memory (GB)	Type	Peak Bandwidth (TBps)
NVIDIA B300 SXM	288	HBM3e	8.00
AMD MI355X	288	HBM3e	8.00
AMD MI350X	288	HBM3e	8.00
AMD MI325X	256	HBM3e	6.00
AMD MI300X	192	HBM3	5.30
NVIDIA GB200 SXM	186	HBM3e	8.00
NVIDIA B200 SXM	180	HBM3e	8.00
NVIDIA GH200 SXM (2)	141	HBM3e	4.80
NVIDIA H200 SXM	141	HBM3e	4.80
Intel Gaudi3	128	HBM2e	3.70
AMD MI250	128	HBM2e	3.28
AMD MI250X	128	HBM2e	3.28
NVIDIA GH200 SXM (1)	96	HBM3	4.00
Intel Gaudi2	96	HBM2e	2.46
AWS Trainium2 / Ultra	96	HBM3	2.90
Google TPU v5p	95	HBM2e	2.76
NVIDIA H100 SXM	80	HBM3	3.35
NVIDIA A100 SXM	80	HBM2e	2.00
NVIDIA H100 PCIe	80	HBM3	2.00

Accelerator	Memory (GB)	Type	Peak Bandwidth (TBps)
NVIDIA A100 PCIe	80	HBM2e	1.94
NVIDIA L40S	48	GDDR6	0.86
Google TPU v4	32	HBM2	1.20
Google TPU v5e	16	HBM2	0.82

Announced, availability not confirmed:

Accelerator	Memory (GB)	Type	Peak Bandwidth (TBps)
AMD MI455X	432	HBM4	23.30
NVIDIA Rubin SXM	288	HBM4	22.00
Huawei Ascend 950DT	96	?	4.00

Notes:

- The listed sizes preserve vendor-published `GB` labels. As of 2026-07-29, NVIDIA's HGX AI Factory components page lists B200 at 180GB, and its MIG table lists B200 at 180GB and GB200 at 186GB. NVIDIA's OpenFold2 support matrix instead lists B200 at 192GB without explaining whether the discrepancy reflects physical, usable, reserved, or differently reported capacity. This table follows the hardware and platform values; the possible physical-capacity/ECC explanation above is an arithmetic derivation, not a documented NVIDIA specification.
- I didn't include `NVIDIA H100 dual NVL` as it's 2x H100 GPUs with 14GiB memory extra per chip and slightly faster memory (3.9TBps vs 3.35TBps) - but it would have an unfair advantage in the above table as everything else is per-chip. (I guess AMD250 is also 2 GCDs, but they aren't very competitive anyway and will soon be displaced from this table by newer offerings)

How to calculate theoretical memory bandwidth

The `Peak Bandwidth` figures above aren't measured - like the theoretical TFLOPS they fall out of three numbers:

```
bandwidth (bytes/sec) = memory clock (Hz) * bus width (bytes) * 2
```

The `*2` is because HBM, like all Double Data Rate memory, transfers on both the rising and the falling clock edge - hence one transfer per half-cycle rather than per cycle. The bus width is the *whole* memory interface summed across every stack, not one stack's width, which is the easiest of the three to get wrong.

Worked on a B200: `nvidia-smi -q -d CLOCK` reports a 3996MHz memory clock and the bus is 8192 bits, so 1024 bytes:

- `3996*10**6 * 1024 * 2 / 10**12 = 8.184` TBps

against the 8.00TBps NVIDIA publishes and this table carries. The 2.3% gap is the published figure being rounded down,

not an error in the arithmetic - which is the normal outcome and the reason this is worth doing as a *cross-check* rather than as a way to obtain the number. If your result is far off rather than a couple of percent off, one of the three inputs is wrong.

All three inputs are readable from the accelerator itself, so nothing has to be transcribed from a spec sheet:

```
import torch
p = torch.cuda.get_device_properties(0)
# memory_clock_rate is the *peak* clock in kHz; memory_bus_width is in bits
bw = p.memory_clock_rate * 1e3 * (p.memory_bus_width / 8) * 2 / 1e12
print(f"{p.name}: {p.memory_clock_rate/1e6:.3f}GHz x {p.memory_bus_width}-bit bus ->
{bw:.2f}TBps")
```

Two things to watch. Take the **peak** memory clock rather than the current one - an idle accelerator clocks its memory down, so `nvidia-smi`'s `Clocks` section will read low while `Max Clocks` reads the figure this arithmetic needs, and PyTorch's `memory_clock_rate` is already the peak. And this gives the *theoretical* ceiling; what a real workload achieves is lower, which is what Maximum Achievable Matmul FLOPS is to TFLOPS - see Do more SMs give more TFLOPS? for why aggregate peaks and achieved throughput diverge in general.

Memory speed (bandwidth) is, of course, very important since if it's not fast enough, the compute ends up idling waiting for the data to be moved to and from the memory.

Caches

High performance cache is used for storing frequently used instructions and data. L1 is usually the smallest and fastest, then L2 is a bit larger and a bit slower and there can be an L3 cache which is even bigger and slower. All these caches massively reduce trips to HBM.

The cache size is often important for when running benchmarks - as one needs to reset the cache between each experiment.

Cache specifications are difficult to normalize across vendors because they describe different resources and reporting scopes. For example, NVIDIA reports a combined and configurable L1/texture/shared-memory capacity per SM, AMD reports L1 per CU and L2 per XCD, and Intel Gaudi includes software-managed SRAM and configurable cache modes that don't map directly to a conventional GPU cache hierarchy.

Therefore the following tables preserve the scope used in the vendor documentation and don't derive per-accelerator totals by multiplying private local caches. Accelerators without a directly documented comparable value are omitted rather than filled with an estimate.

The first table compares the capacity of broadly shared on-chip caches. It is sorted by **Shared capacity** descending, then by **Approx. announced** descending. For a grouped row, the date is that of the newest product named in the row. This helps answer which accelerator publishes more broadly shared cache capacity, but it isn't a cache-performance ranking: cache type, bandwidth, latency, sharing domain, and workload hit rate still differ. Vendor unit labels are preserved, so the ordering of `MB` and `MiB` values is approximate.

Shared capacity	Accelerator / architecture	Cache type	Scope / qualification	Approx. announced	Ref.
256MiB	AMD MI350X / MI355X	Infinity Cache (LLC)	per accelerator	2025-06	1
256MiB	AMD MI300X / MI325X	Infinity Cache (LLC)	per accelerator	2024-10	2
192MB	AMD MI455X	global L2	per accelerator	2026-06	3
126MB	NVIDIA Blackwell (GB200	L2	per GPU	2024-03	4

Shared capacity	Accelerator / architecture	Cache type	Scope / qualification	Approx. announced	Ref.
	example)				
96MiB	Intel Gaudi3	configurable L3 or 4 L2 slices	per accelerator	2024-04	5
50MiB	NVIDIA Hopper (H100 example)	L2	per accelerator	2022-03	6
40MiB	NVIDIA Ampere (A100 example)	L2	per accelerator	2020-05	7

Sources:

1. AMD MI350X / MI355X cache specification and announcement
2. AMD MI300X / MI325X cache specification and announcement
3. AMD Instinct MI455X specifications and announcement
4. NVIDIA Blackwell cache specification and announcement
5. Intel Gaudi3 cache specification and announcement
6. NVIDIA Hopper cache specification and announcement
7. NVIDIA Ampere cache specification and announcement

The second table preserves vendor-native local resources. It is sorted by **Vendor** ascending, then by **Approx. announced** descending.

Vendor	Accelerator / architecture	Approx. announced	Vendor-reported local resource	Scope	Ref.
AMD	MI455X	2026-06	not disclosed in the cited public specification		1
AMD	MI350X / MI355X	2025-06	32KiB L1 data cache; 4MiB L2 cache	per CU; per XCD	2
AMD	MI300X / MI325X	2024-10	32KiB L1 data cache; 4MiB L2 cache	per CU; per XCD	3
Intel	Gaudi3	2024-04	24MiB configurable cache	per DCORE; 4 DCOREs	4
Intel	Gaudi2	2022-05	48MiB software-managed SRAM	per accelerator	5
NVIDIA	Blackwell (GB200 example)	2024-03	256KB combined L1/texture/shared memory	per SM	6
NVIDIA	Hopper (H100 example)	2022-03	256KiB combined L1/texture/shared memory	per SM	7
NVIDIA	Ampere (A100 example)	2020-05	192KiB combined L1/texture/shared memory	per SM	8

Sources:

1. AMD Instinct MI455X specifications and announcement
2. AMD MI350X / MI355X cache specification and announcement

3. AMD MI300X / MI325X cache specification and announcement
4. Intel Gaudi3 cache specification and announcement
5. Intel Gaudi2 architecture specification and announcement
6. NVIDIA Blackwell cache specification and announcement
7. NVIDIA Hopper cache specification and announcement
8. NVIDIA Ampere cache specification and announcement

The third table restores the per-product detail from the earlier comparison while separating vendor-native scope from derived distributed capacity. It is sorted by **Vendor** ascending, then by **Accelerator / product** ascending. `Distributed capacity` is arithmetic across private units and does not imply that one operation can access it as a single shared cache.

Vendor	Accelerator / product	Vendor-native detail	Distributed capacity	Shared cache / LLC	Qualification of the earlier entry	Ref.
AMD	MI300X	32KiB L1/CU; 4MiB L2/XCD; 8 XCDs	32MiB L2; L1 not derived	256MiB Infinity Cache	Earlier 0.25MiB L1 total multiplied a per-CU value by the eight-XCD count	1
AMD	MI325X	32KiB L1/CU; 4MiB L2/XCD; 8 XCDs	32MiB L2; L1 not derived	256MiB Infinity Cache	Earlier 0.25MiB L1 total multiplied a per-CU value by the eight-XCD count	1
AMD	MI355X	32KiB L1/CU; 4MiB L2/XCD; 8 XCDs	32MiB L2; L1 not derived	256MiB Infinity Cache	Earlier 0.25MiB L1 total multiplied a per-CU value by the eight-XCD count	1
Intel	Gaudi2	48MiB software-managed SRAM per accelerator	not applicable	no conventional GPU LLC	Earlier 48MiB L2 label is retained here with its software-managed meaning	2
Intel	Gaudi3	24MiB configurable cache/DCORE; 4 DCOREs	96MiB	configurable L3 or 4 L2s	Earlier 24MiB x 4 = 96MiB arithmetic is retained with its cache mode	3
NVIDIA	A100 PCIe	192KiB combined L1/ texture/shared memory/SM; 108 SMs	20.25MiB	40MiB L2	Earlier row said 128KiB/SM, but its 20.25MiB total used 192KiB/SM	4
NVIDIA	A100 SXM	192KiB combined L1/ texture/shared memory/SM; 108 SMs	20.25MiB	40MiB L2	Earlier row said 128KiB/SM, but its 20.25MiB total used 192KiB/SM	4
NVIDIA	B200 SXM	not disclosed in the cited product-specific form	not derived	not disclosed	Earlier `???` placeholder is retained as `not disclosed`	5
NVIDIA	B300 SXM	not disclosed in the cited product-specific form	not derived	not disclosed	Earlier `???` placeholder is retained as `not disclosed`	5
NVIDIA	GH100 full implementation	256KiB combined L1/ texture/shared memory/SM; 144 SMs	36MiB	up to 60MB L2	Earlier `GH100 SXM` mixed 132 enabled SMs with the full-die 60MB L2	6, 7
NVIDIA	GH200 SXM (original label)	uses its installed Hopper GPU cache	not derived	use the GPU configuration	Earlier 256KiB x 132 = 33MiB and 60MiB L2 mixed	6, 7

Vendor	Accelerator / product	Vendor-native detail	Distributed capacity	Shared cache / LLC	Qualification of the earlier entry	Ref.
		hierarchy			product/full-die scopes	
NVIDIA	H100 SXM	256KiB combined L1/ texture/shared memory/SM; 132 SMs	33MiB	50MiB L2	Earlier 192KiB x 132 = 24.75MiB distributed-capacity value is preserved	6, 7
NVIDIA	H200 SXM	256KiB combined L1/ texture/shared memory/SM; 132 SMs	33MiB	50MiB L2	Earlier 192KiB x 132 = 24.75MiB distributed-capacity value is preserved	6, 7

Sources:

1. AMD Instinct MI300 Series / MI350 Series workload optimization
2. Intel Gaudi architecture
3. Intel Gaudi3 white paper
4. NVIDIA Ampere Tuning Guide
5. NVIDIA Blackwell Tuning Guide
6. NVIDIA Hopper Tuning Guide
7. NVIDIA Hopper architecture white paper

When comparing these specifications:

- First compare the scope. A per-SM or per-CU cache is private to a local compute unit, whereas an L2, L3, LLC, or Infinity Cache may be shared by a much larger part of the accelerator.
- Check the semantics. A hardware-managed cache, configurable shared memory, and software-managed SRAM may have similar capacities while requiring very different programming and access patterns.
- Don't treat the sum of private caches as one large shared cache. Such a total describes physical capacity, but not how much cache any one operation can access.
- Capacity is only one factor. Cache bandwidth, latency, associativity, sharing domain, and the workload's hit rate can matter more, so use workload-representative benchmarks for performance comparisons.

Notes:

1. AMD provides L3 AMD Infinity Cache which it also calls Last Level Cache (LLC) in the specs
2. Gaudi has a different architecture than a GPU. In Gaudi's case, the MME and TPC have private buffers that perform some of the functions of an L1 cache, called Suspension Buffers. The main function that these buffers provide is data reuse from the buffer instead of reading the same data multiple times from L2/L3/ HBM. Both Gaudi2 and Gaudi3 have the same Suspension Buffers for the TPC and MME.
3. Gaudi2 doesn't have a conventional GPU cache hierarchy. It has scratchpad SRAM, meaning that software determines what goes in or out of the SRAM. There are dedicated DMA engines that software needs to program to perform the data movement between SRAM and HBM.
4. Gaudi3's 96MiB cache can be configured by software to be either a single L3 cache or 4 slices of 24MiB L2 cache (this is at tensor-level granularity). L2 configuration is 2x faster than L3.

Clock speed

Also known as clock rate this spec tells us at which frequency the card runs. As hardware becomes faster newer generations will typically increase the clock speed.

Accelerator specifications may publish a base clock, a boost or peak clock, or no compute clock at all:

- A base clock is a vendor-defined nominal operating point, not the minimum idle clock.
- A boost, peak, or maximum clock is a vendor-defined upper operating target. It is not guaranteed under every workload because the actual frequency depends on the accelerator SKU, power limit, temperature, and workload.

Clock frequency is useful for calculating theoretical TFLOPS within an architecture, but raw MHz alone does not rank performance across different architectures.

The same accelerator family may also have different clock rates across SKUs and system configurations, so always check the clock of your specific accelerator.

The table is sorted by compute clock, highest first. Products whose vendors do not disclose a compute clock follow alphabetically. `not disclosed` means that the linked product specification did not publish a compute clock when checked on 2026-07-28.

Generally available:

Accelerator	Compute Clock (MHz)	Notes
AMD MI355X	2400	1; Peak Engine Clock
AMD MI300X	2100	3; Peak Engine Clock
AMD MI325X	2100	4; Peak Engine Clock
NVIDIA GB200 SXM	2062	5; device-reported maximum SM clock
NVIDIA B200 SXM	1965	5; device-reported maximum SM clock
NVIDIA H200 SXM	1830	12
NVIDIA H100 SXM	1830	12
Intel Gaudi2	1650	6,10,11; device-reported MME clock; TPC=1800
Intel Gaudi3	1600	6,10,11; device-reported MME clock; TPC=1600
NVIDIA A100 SXM	1410	7; GPU Boost Clock
NVIDIA A100 PCIe	1410	7; GPU Boost Clock
NVIDIA B300 SXM	not disclosed	8
NVIDIA GB300 SXM	not disclosed	8

Announced, availability not confirmed:

Accelerator	Compute Clock (MHz)	Notes
AMD MI455X	2400	2; Peak Engine Clock
NVIDIA Rubin GPU	not disclosed	9

Notes:

1. AMD Instinct MI355X specifications

2. AMD Instinct MI455X specifications
3. AMD Instinct MI300X specifications
4. AMD Instinct MI325X specifications
5. NVIDIA System Management Interface documentation
6. Intel Gaudi `hl-smi` documentation
7. NVIDIA A100 architecture white paper
8. NVIDIA GB300 NVL72 specifications
9. NVIDIA Vera Rubin NVL72 specifications
10. Intel Gaudi architecture documentation
11. Intel Gaudi exposes separate clocks for its Matrix Multiplication Engine (MME) and Tensor Processing Core (TPC).
12. 1830MHz is NVIDIA's official compute clock for H200 and H100, superseding the 1980MHz that was widely published earlier. It is also self-consistent: it reproduces the 989TFLOPS bf16 spec exactly through the TFLOPS calculation, where 1980MHz would give 1070.

Here is how to get the actual clock speed (in particular when your accelerator is under load):

- NVIDIA: `nvidia-settings -q GPUCurrentClockFreqs` with X-server, or `nvidia-smi -q -d CLOCK -i 0 | grep -B2 " SM " | head -3` for headless (adapt `-i 0` if not measuring gpu0, or remove it to show all available GPUs). Remove the `grep` to get the full output - `Max Clocks` shows the theoretical clock. Here is a continuous log of the same with timestamps: `nvidia-smi --query-gpu=index,timestamp,power.draw,clocks.sm,clocks.mem,clocks.gr --format=csv -l 1 -i 0`
- AMD: `rocm-smi -g` for actual and `amd-smi metric --clock` for theoretical
- Intel: `hl-smi -Q clocks.current.soc --format=csv` for actual and `hl-smi -Q clocks.max.soc --format=csv` or `hl-smi -Q clocks.limit.soc --format=csv` for theoretical

Power consumption

Accelerator vendors publish power specifications under several names: Thermal Design Power or Thermal Design Parameter (TDP), Total Graphics Power (TGP), and Typical Board Power or Total Board Power (TBP).

These terms are related because each describes a power or thermal envelope used to provision electricity and cooling. The underlying physical scopes can be nested: chip or package, accelerator board or module, then complete system. A broader scope includes more components, but the acronym alone does not reliably identify that scope. TDP, TGP, and TBP therefore cannot be universally ordered or converted.

Where vendor documentation defines the scope and qualifier, the relationship is useful:

- NVIDIA defines TGP as the total electrical power for the GPU core, onboard memory, and supporting board circuitry. Its DGX B200 power controls separately expose a 1000W maximum TGP, a 700W default set point, the configured set point, and current measured power.
- AMD lists MI250X with 500W TDP and 560W peak TBP, so for this OAM module the published peak TBP is 60W above its TDP.
- NVIDIA's GB300 rack-planning example adds four 1.4kW GPUs and a 1kW CPU into a 6.6kW managed node budget, adds 0.9kW of static node power, then multiplies the resulting 7.5kW node budget by 18 nodes to obtain a 135kW rack budget.

These examples are product-specific relationships, not conversion rules. Before comparing values, check whether the specification describes electrical input or thermal design, which components it includes, and whether the value is typical, default, maximum, peak, or configurable. Actual draw varies with workload and the configured power limit.

If you're a cloud compute user you normally don't care about these values because power consumption is already included in your package. For those who host their own hardware, these values help determine how much power and cooling to provision without power-limit or thermal throttling.

Beyond sizing power and cooling infrastructure, these specifications can also help explain sustained performance. With

adequate cooling, more power headroom may reduce power-limit throttling and allow an accelerator to sustain higher performance, but a higher power specification alone does not prove faster compute or better efficiency. Efficiency requires comparable work-per-energy measurements such as FLOP/J or tokens/J. For example, AMD publishes the same peak arithmetic throughput for MI325X and MI300X, but MI325X pairs a 1000W rather than 750W peak TBP with 33% more HBM capacity and 13% more memory bandwidth; its nominal peak arithmetic per watt is therefore 25% lower, while measured efficiency may differ on memory-bound workloads.

The table is sorted by **Power spec (W)**, highest first, followed by `N/A` entries. Values are per accelerator rather than per node or rack. `N/A` means that the cited specification did not publish a product power value when checked on 2026-07-29.

Generally available:

Accelerator	Power spec (W)	Vendor term	Notes
NVIDIA GB300 SXM	1400	per-GPU maximum power	1
AMD MI355X	1400	Typical Board Power	2
NVIDIA GB200 SXM	1200	per-GPU maximum power	1
AMD MI325X	1000	Typical Board Power	3; peak
AMD MI350X	1000	Typical Board Power	4
NVIDIA B200 SXM	1000	maximum TGP	5
NVIDIA B300 SXM	1000	maximum TGP	6
Intel Gaudi3	900	TDP	7
AMD MI300X	750	Typical Board Power	8; peak
NVIDIA H100 SXM	700	maximum TDP	9
NVIDIA H200 SXM	700	maximum TDP	10
AMD MI350P	600	maximum Typical Board Power	11; configurable to 450W
Intel Gaudi2	600	TDP	12
NVIDIA H200 NVL	600	maximum TDP	10
NVIDIA RTX PRO 6000	600	maximum power consumption	13; configurable
AMD MI250X	560	Typical Board Power	14; peak; 500W TDP
NVIDIA H100 NVL	400	maximum TDP	9; configurable 350-400W
NVIDIA A100 SXM	400	maximum TDP	15; CTS up to 500W
NVIDIA L40S	350	maximum power consumption	16
NVIDIA A100 PCIe	300	maximum TDP	15
AWS Trainium2 / Ultra	N/A		18
Google TPUs	N/A		20

Announced, availability not confirmed:

Accelerator	Power spec (W)	Vendor term	Notes
AMD MI455X	N/A		17
NVIDIA Rubin GPU	N/A		19

Notes:

1. NVIDIA Mission Control power-management FAQs
2. AMD Instinct MI355X specifications
3. AMD Instinct MI325X specifications
4. AMD Instinct MI350X specifications
5. NVIDIA DGX B200 power capping
6. NVIDIA DGX B300 power capping
7. Intel Gaudi 3 AI accelerator white paper
8. AMD Instinct MI300X specifications
9. NVIDIA H100 specifications
10. NVIDIA H200 specifications
11. AMD Instinct MI350P specifications
12. Intel Gaudi 2 AI accelerator white paper
13. NVIDIA RTX PRO 6000 Blackwell Server Edition specifications
14. AMD Instinct MI250X specifications
15. NVIDIA A100 specifications
16. NVIDIA L40S specifications
17. AMD Instinct MI455X specifications
18. AWS Trainium2 architecture documentation
19. NVIDIA Vera Rubin NVL72 specifications
20. Google doesn't publish power consumption specs for recent TPUs, the older ones can be found here

Supplying power to your own cards

Some high end consumer GPU cards have 2 and sometimes 3 PCIe 8-Pin power sockets. Make sure you have as many independent 12V PCIe 8-Pin cables plugged into the card as there are sockets. Do not use the 2 splits at one end of the same cable (also known as pigtail cable). That is if you have 2 sockets on the GPU, you want 2 PCIe 8-Pin cables going from your PSU to the card and not one that has 2 PCIe 8-Pin connectors at the end! You won't get the full performance out of your card otherwise.

Each PCIe 8-Pin power cable needs to be plugged into a 12V rail on the PSU side and can supply up to 150W of power.

Some other cards may use a PCIe 12-Pin connectors, and these can deliver up to 500-600W of power.

Low end cards may use 6-Pin connectors, which supply up to 75W of power.

Additionally you want the high-end PSU that has stable voltage. Some lower quality ones may not give the card the stable voltage it needs to function at its peak.

And of course the PSU needs to have enough unused Watts to power the card.

Cooling

The only important practical understanding for cooling is that if the accelerators aren't kept cool they will throttle their compute clock and slow everything down and could even crash sometimes, albeit throttling is supposed to prevent that.

It's hard to tell the exact best temperature to strive for when a GPU is heavily loaded, but probably anything under +80C is good, but lower is better - perhaps 70-75C is an excellent range to be in. The throttling down is likely to start at around 84-90C. But other than throttling performance a prolonged very high temperature is likely to reduce the lifespan of a GPU.

It's important to understand that on modern GPUs throttling is not just a hard cliff at the thermal limit, the boost clock itself is a continuous function of temperature (and power and current). NVIDIA's GPU Boost (and AMD's equivalent on Instinct) continuously samples die temperature, board power, and per-rail current, and picks the highest stable point on the chip's V-F-T curve that fits inside all of those budgets. Cross any one of them and the clock steps down; come back under and it steps back up. In practice this means **the cooler you keep the GPU, the higher the sustained clock you get**, well before you hit the official throttle threshold. A GPU running steady at 60°C will hold a meaningfully higher average clock than the same GPU running at 78°C on the same workload - even though neither is "throttling" in the alarming sense. This is why liquid-cooled nodes typically deliver higher sustained MAMF than air-cooled nodes with identical silicon, and why a node with one poorly-seated heatsink can have one obvious straggler GPU.

For NVIDIA GPUs to check if your GPU gets throttled down, run `nvidia-smi -q -d PERFORMANCE` - if `SW Thermal Slowdown` or some other entries are `Active` - then your are not getting the full performance of your GPU and you need to investigate better cooling.

High end accelerators for ML workloads

Cloud accelerators

Most common accelerators that can be either rented on compute clouds or purchased:

NVIDIA:

- Vera Rubin NVL72 -- the 72 GPUs supernode at NVLink speed with Grace Rubin - 36x blocks of 2x Rubin GPU + Vera CPU
- GB300 NVL72 - the 72 GPUs supernode at NVLink speed with B300s (Grace Blackwell - 36x blocks of 2x B300 + Grace CPU)
- GB200 NVL72 - the 72 GPUs supernode at NVLink speed with B200s. (Grace Blackwell - 36x blocks of 2x B200 + Grace CPU)
- B300 - NVIDIA now publishes official HGX B300 platform specs and lists it as shipping, as of 2026-07-31. Note that those are 8-GPU board figures (144PFLOPS FP4 sparse, 2.1TB total memory, 14.4TBps total NVLink) so per-GPU values have to be divided out, and HBM bandwidth still isn't published there - for that the DGX B300 datasheet remains the practical source
- B200 - same story: official HGX B200 platform specs published and shipping (144PFLOPS FP4 sparse, 1.4TB total memory, 14.4TBps total NVLink), with the DGX B200 datasheet filling in what the platform page omits. Worth knowing that B300 cut FP64 hard - 10TFLOPS against B200's 296TFLOPS per the same table - so it is not a strict upgrade for double-precision work
- H200 - mainly the same as H100, but with more and faster memory! Widely available as of 2026-07-31.
- H100 - 2-3x faster than A100 (half precision), 6x faster for fp8, has been available on all Tier-1 compute clouds since Q4-2023.
- GH200 - 2 chips on one card - (1) H100 w/ 96GiB HBM3 or 144GiB HBM3e + (2) Grace CPU w/ 624GiB RAM - available as of 2026-07-31. Do not confuse with H200, which is a different card.
- L40S - a powerful card that is supposed to be more than 2x cheaper than H100, and it's more powerful than A100.
- A100 - huge availability, but already getting outdated. But given the much lower cost than H100 this is still a great GPU.

AMD:

- MI455X ~= Rubin, a little above it on the published numbers - but those are AMD Performance Labs projections carrying "Results subject to change when products are released in market", and it isn't

purchasable as of 2026-07-31. See the TFLOPS and memory tables, where it sits in the `Announced, availability not confirmed` half

- MI355X ~= B200 - just starting to emerge, mainly on Tier-2 clouds
- MI350X ~= B200 - it seems that MI355X is made available instead of MI350X
- MI325X ~= H200 - available mainly on Tier-2 clouds
- MI300X ~= H100 - available mainly on Tier-2 clouds (lots of new startups)
- MI250 ~= A100 - very few clouds have them

Intel:

- Falcon Shores - canceled
- Gaudi3, somewhat below B200 theoretical TFLOPS-wise - already available on Intel cloud - spec
- Crescent Island - inference GPU, sampling H2-2026; 160GB LPDDR5X / 350W, no published TFLOPS - Intel Crescent Island
- Gaudi2 somewhere between A100 and H100 theoretical TFLOPS-wise spec - available on Intel cloud. AWS has the older Gaudi1 via DL1 instances. It's also available on-premises implementations via Supermicro and WiWynn.

Amazon:

- Trainium2 < H100 - available on AWS (works via PyTorch XLA)

Google:

- TPU7x (Ironwood) - 2307TFLOPS bf16 and 192GB HBM per chip, in 4-chip VMs up to a 9216-chip pod. Rent-only on GCP, like every TPU generation
- TPU v6e (Trillium) and TPU v5p - the previous two generations, both still offered

SambaNova:

- SN40L - fourth-generation RDU, 16 per SambaRack; inference-oriented rather than a training part
- SN50 - fifth generation, announced Feb-2026, "will start shipping to customers in the second half of 2026"

Cerebras on-premises clusters

Cerebras cluster and systems based on WaferScale Engine (WSE).

Cloud-only solutions

These can be only used via clouds:

Google TPUs, specs - lock-in, can't switch to another vendor like NVIDIA -> AMD

Recent architecture specs:

- v7x
- v6e
- v5p

Cerebras:

- Cloud

New hardware startups

These are possible future competitors to the big boys.

They typically target inference.

- TensTorrent, n150s/n300s specs
- d-Matrix, specs

How to get the best price

Remember that the advertised prices are almost always open to negotiations as long as you're willing to buy/rent in bulk or if renting for a 1-3 years. What you will discover is that the actual price that you end up paying could be many times less than the advertised "public" price. Some cloud providers already include the discount as you choose a longer commitment on their website, but it's always the best to negotiate directly with their sales team. In addition or instead of a $$-discount you could be offered some useful features/upgrades for free.

If your company has venture capital investors - it could help a lot to mention that, as then the cloud provider knows you are likely to buy more compute down the road and more likely to discount more.

Tier 2 clouds are likely to give better prices than Tier 1. Tier 1 as of 2026-08 is AWS, OCI, Azure and GCP.

For the baseline prices it should be easy to find a few good sites that provide an up-to-date public price comparisons across clouds - just search for something like cloud gpu pricing comparison. Some good starting points: vast.ai and specifically for clusters gpulist.ai.

When shopping for a solution please remember that it's not enough to rent the most powerful accelerator. You also need fast intra-node and inter-node connectivity and sufficiently fast storage - without which the expensive accelerators will idle waiting for data to arrive and you could be wasting a lot money and losing time.

Accelerators in detail

NVIDIA

Abbreviations:

- CUDA: Compute Unified Device Architecture (proprietary to NVIDIA)

NVIDIA-specific key GPU characteristics:

- CUDA Cores - similar to CPU cores, but unlike CPUs that typically have 10-100 powerful cores, CUDA Cores are weaker and come in thousands and allow to perform massive general purpose computations (parallelization). Like CPU cores CUDA Cores perform a single operation in each clock cycle.
- Tensor Cores - special compute units that are designed specifically to perform fast multiplication and addition operations like matrix multiplication. These perform multiple operations in each clock cycle. They can execute extremely fast computations on low or mixed precision data types with some loss (fp16, bf16, tf32, fp8, etc.). These cores are specifically designed for ML workloads.
- Streaming Multiprocessors (SM) are clusters of CUDA Cores, Tensor Cores and other components.

For example, A100-80GB has:

- 6912 CUDA Cores
- 432 Tensor Cores (Gen 3)
- 108 Streaming Multiprocessors (SM)

H100 has:

- 16896 FP32 CUDA Cores
- 528 Tensor Cores (Gen 4)
- 132 Streaming Multiprocessors (SM)

AMD

AMD-specific key GPU characteristics:

- Stream Processors - are similar in functionality to CUDA Cores - that is these are the parallel computation units. But they aren't the same, so one can't compare 2 gpus by just comparing the number of CUDA Cores vs the number of Stream Processors.

- Compute Units - are clusters of Stream Processors and other components

for example, AMD MI250 has:

- 13,312 Stream Processors
- 208 Compute Units

AMD's table comparing its high-end gpus

Intel Gaudi

Architecture

- 24x 100 Gigabit Ethernet (RoCEv2) integrated on chip - 21 of which are used for intra-node and 3 for inter-node (so `21*8=168` cards for intra-node (262.5GBps per Gaudi chip), and `3*8=24` cards for inter-node (2.4Tbps between nodes)
- 96GiB HBM2E memory on board w/2.45TBps bandwidth per chip, for a total of 768GiB per node

A server/node is built from 8 Gaudi accelerators, which can then be expanded with racks of those servers.

There are no official TFLOPS information published (and from talking to an Intel representative they have no intention to publish any.) They publish the following benchmarks but I'm not sure how these can be used to compare this compute to other providers.

Comparison: supposedly Gaudi2 competes with NVIDIA H100

On what comes after Gaudi: Falcon Shores was to replace it and was canceled. Gaudi3 remains Intel's current training-class accelerator. The announced next data-center GPU is Crescent Island.

Intel Crescent Island

Crescent Island is an inference-oriented Xe3P GPU with customer sampling in H2-2026. Intel's card is 160GB LPDDR5X (ODMs up to 480GB), 350W air-cooled PCIe; Intel has not published TFLOPS. OCP announcement, Hot Chips 2026

AWS Trainium

AWS-specific vocabulary:

- NeuronCore - the compute unit. A Trainium2 chip holds "eight NeuronCore-V3 cores"
- NeuronLink - the chip-to-chip interconnect. Trainium2 uses "NeuronLink-v3 for chip-to-chip interconnect provides 1.28 TB/sec bandwidth per chip"
- CC-Cores - dedicated collective-communication engines; "16 CC-Cores orchestrate collective communication among Trainium2 chips within and across instances"

A `trn2.48xlarge` instance holds 16 Trainium2 chips, and an UltraServer extends NeuronLink across "64 Trainium2 chips across four Trn2 instances". Memory pooling spans "Up to 64 chips", against 16 for the previous generation.

You program it through the Neuron SDK and PyTorch XLA rather than CUDA, which is the main porting cost. AWS publishes no per-link rate or link count, only the 1.28TBps aggregate, so peer-to-peer bandwidth can't be derived - see the intra-node tables.

Specs: Trainium2, Trn2 instances.

Google TPU

TPU-specific vocabulary:

- TensorCore and SparseCore - the compute units. A TPU7x chip has "2 TensorCores and 4 SparseCores", split across two chiplets joined by a die-to-die interface
- ICI (inter-chip interconnect) - the scale-up fabric, a 3D torus where each chip has "a direct connection to

the nearest neighboring chips in 3 dimensions"
- slice, cube, pod - the units of allocation; a slice is "a collection of chips all located inside the same TPU Pod connected by high-speed inter-chip interconnects (ICI)"

TPU7x (Ironwood) per chip: 192GB HBM at "approximately 7.37 TB/s", 2307TFLOPS bf16 and 4614TFLOPS fp8, and a "Bidirectional inter-chip interconnect (ICI) bandwidth per chip (GBps)" of 1200. The torus carries "bi-directional bandwidth of 200 GBps per axis", which is consistent with 1200 total across six neighbors. A VM is always 4 chips, and a pod reaches "a 9,216-chip footprint".

The catch isn't performance, it is lock-in: TPUs are rent-only on GCP and the software path is XLA, so a codebase that reaches for custom CUDA kernels doesn't move over for free.

Specs: TPU7x, TPU system architecture.

Huawei Ascend

The one vendor here that most readers can't buy from, and the one with the least accessible documentation - but it ships at scale, so it belongs for context if nothing else.

Huawei-specific vocabulary:

- Ascend - the accelerator line; Atlas - the servers and rack-scale SuperPoDs built from them
- UB Link (UnifiedBus) - the scale-up interconnect, Huawei's answer to NVLink and UALink
- HiF8 - Huawei's own 8-bit float format, quoted alongside the OCP `mxFP8` standard rather than instead of it
- CANN - the compute architecture and toolchain, with MindSpore as the first-party framework and PyTorch/TensorFlow adaptation layers alongside it

As of 2026-08 Huawei's Chinese pages carry the full specification table where the English ones carry almost nothing, so use those. For the Atlas 950 SuperPoD the Chinese page gives up to 1024 Ascend 950DT accelerators, on-chip memory of up to 1024 x 96GB at a bandwidth of up to 4.0TBps, AI compute of up to 1EFLOPS mxFP8/FP8/HiF8 and 2EFLOPS mxFP4, and a per-cabinet total interconnect bandwidth of up to 64 x 1.68TBps bidirectional.

Divided out over the 1024 accelerators, that is roughly 977TFLOPS mxFP8 and 1953TFLOPS mxFP4 per Ascend 950DT, with 96GB at 4.0TBps and 840GBps of unidirectional UB Link scale-up bandwidth each - which would place it just below NVLink 5. Unlike the English page, this division is safe rather than inferred: the accelerator count is stated, and it cross-checks twice over - 1024 accelerators across the stated 16 compute cabinets is 64 per cabinet exactly as the bandwidth line says, and 1024 x 1.68TBps is the 1.72PBps total the same page claims.

Two things to be careful about. The TFLOPS and memory comparison-table rows for Ascend 950DT use the **64-card** Atlas 950 SuperPoD product-spec table on the English Ascend Community page (`114.1 / 58.8 / 31.1` PFLOPS @ mxFP4 / mxFP8-HiF8 / FP16-BF16, and `64 × 96GB` on-chip memory at 4.0TBps) - divided per accelerator - not the 1024-card SuperPoD totals on the Chinese page (~1953 / 977 mxFP4/mxFP8, no bf16). The two configurations are not simply the same part scaled; don't mix figures between them. Memory type is `?` in the table because Huawei publishes "On-chip Memory" / 片上内存 without an HBM generation. And `HiF8` is quoted alongside `mxFP8` as one published rate, so the `fp8` column carries that pair - HiF8 itself is tapered-precision (arxiv:2409.16626), not an OCP `E4M3`/`E5M2`, and the chapter's dtype decoder does not apply to it.

Specs: Ascend Community. Be aware that both Huawei sites are JavaScript applications that serve only a page shell to a plain downloader - `e.huawei.com` returns 255KB of HTML containing not one occurrence of `PFLOPS`, `TB/s` or even `Ascend`. The Ascend Community site happens to inline enough text to read the Atlas 950 spec off; the product pages do not. So these figures are harder to confirm than any other vendor's here, and a second-hand UB Link or Ascend number should be treated as unverified until traced back to Huawei.

Cerebras

The outlier in approach: rather than connecting many chips, put the whole cluster on one wafer. WSE-3 measures "46,225 mm2 and containing 4 trillion transistors", delivers compute through "900,000 AI-optimized cores", and is rated at "125

petaflops of AI compute".

Wafer-scale trades a familiar problem for an unfamiliar one. There is no intra-node fabric to benchmark and no model-parallel partitioning across accelerators, because it is one device - but you also can't buy 8 of them in a node the way the rest of this chapter assumes, and the software stack is entirely its own. Yield is handled architecturally rather than avoided, "with redundant compute cores, redundant routing, and a fail-in-place architecture".

On-chip SRAM capacity and memory bandwidth are not published on the chip page despite a heading promising them, so the two figures that matter most for a training workload have to come from the datasheet.

Specs: Cerebras WSE.

SambaNova

Inference-first, and explicitly so - the vocabulary and the product line are both organized around serving rather than training.

SambaNova-specific vocabulary:

- RDU (Reconfigurable Dataflow Unit) - the accelerator. The "fourth-generation RDU SN40" and "fifth-generation SN50" are the current parts
- Dataflow architecture - the execution model, reconfiguring the chip around a model's graph instead of dispatching kernels
- SambaRack - the rack; "The combination of 16 SN40L RDUs creates a single, high-performance rack"
- three-tier memory - HBM plus large-capacity memory plus SRAM, so that "Models residing in HBM and SRAM can be hot swapped in milliseconds"

SN50 was announced in February 2026 as the fifth generation and "will start shipping to customers in the second half of 2026", so SN40L is what you can actually deploy today. A SambaRack SN50 combines 16 chips for "five times more compute per accelerator and four times more network bandwidth", scales "up to 256 chips across multiple racks", and targets models "up to 10 trillion parameters in size" at context lengths "up to 10 million tokens".

The three-tier memory and millisecond model swapping are the genuinely different idea here - it is aimed at serving many models from one machine, which is a different problem from the one the rest of this chapter optimises for. Don't read SambaNova's throughput comparisons against B200 as training numbers; they are agentic-inference numbers.

Specs: RDU AI chips, SambaRack.

API

Which software is needed to deploy the high end accelerators?

NVIDIA

NVIDIA GPUs run on CUDA

AMD

AMD GPUs run on ROCm - note that PyTorch intentionally reuses the same `torch.cuda` Python API on ROCm, so many programs that only use PyTorch APIs can run unchanged on supported AMD GPUs such as MI250 and MI300X! Software that includes custom CUDA or PTX code, NVIDIA-specific libraries, or unsupported operations may still need porting or replacement and validation.

Intel Gaudi

The API is via Habana SynapseAI® SDK which supports PyTorch and TensorFlow.

Useful integrations:

- HF Optimum Habana which also includes - DeepSpeed integration.

Apples-to-apples Comparison

It's very difficult to compare specs of different offerings since marketing tricks get deployed pretty much by all competitors so that one can't compare 2 sets of specs and know the actual difference.

- MLPerf via MLCommons publishes various hardware benchmarks that measure training, inference, storage and other tasks' performance. The round numbers advance a few times a year, so head to the suite pages rather than a pinned version - training and inference: datacenter each show the latest results.

  Except I have no idea how to make use of it - it's close to impossible to make sense of or control the view. This is a great intention lost in over-engineering and not thinking about how the user will benefit from it, IMHO. For example, I don't care about CV data, I only want to quickly see the LLM rows, but I can't do it. And then the comparisons are still not apples to apples so how can you possibly make sense of which hardware is better I don't know.

Accelerator Benchmarks

How to benchmark accelerators

CUDA benchmarks

There are a few excellent detailed write ups on how to perform CUDA benchmarks:

1. How to Accurately Time CUDA Kernels in PyTorch
2. How to Benchmark Code on CUDA Devices? - this one is different from (1) in that it suggests to set both GPU and Memory clocks, whereas (1) only locks the GPU clock.

You can see these instructions applied in mamf-finder.py (other than clock locking)

Input data affects measured performance

Kernel time is not a function of tensor shapes alone. The *values* flowing through a GEMM change how hard the chip works, and that feeds back into the clock.

Dynamic power is roughly proportional to clock frequency times the number of transistors that flip. High-entropy or "unpredictable" bit patterns flip more transistors per cycle than sparse, sorted, or all-zero patterns, so the same-shape matmul draws more power. Once draw approaches the configured power (or thermal / current) limit, NVIDIA's GPU Boost and AMD's equivalent step the SM clock down to stay inside the envelope - the mechanism already described under Power consumption, Cooling and the chip's V-F-T curve. So less "predictable" inputs can make an identical kernel run slower, without any change in the code or the shapes. Horace He's Strangely, Matrix Multiplications on GPUs Run Faster When Given "Predictable" Data demonstrates the effect by comparing all-zeros against random inputs; Input-Dependent Power Usage in GPUs measures GEMM power swinging by up to ~38-40% across input patterns (entropy, sparsity, bit similarity, Hamming weight). A secondary, smaller path is denormals / subnormals: some ops take a slower route for them, and the reduced normal range of bf16/fp16 makes them more likely - see NVIDIA's Flush Denormals with Confidence.

Practical consequences for a benchmark:

1. **Seed the RNG that builds the inputs** (`torch.manual_seed(...)` before the tensors are created). Without a fixed seed, run-to-run input variation adds a small, uncontrolled jitter on top of measurement noise, even when the *distribution* is unchanged.
2. **Use a realistic distribution**, not all zeros or other pathological patterns. All-zeros under-reports power and over-reports throughput - exactly the benchmarking trap Horace's post is about. Uniform-random token IDs (or whatever your real workload feeds) keep the measured power / clock regime close to production.
3. **Expect a small effect when the distribution is held fixed**, not zero. Seeding removes one noise source; it does not make two differently-valued tensors of the same shape run at identical clocks.

This is distinct from numerical reproducibility, which forces the *same results* via deterministic algorithms. Here the goal is the *same timing conditions* - same data, same power draw, same clock.

Maximum Achievable and Sustainable Matmul FLOPS finder

Maximum Achievable Matmul FLOPS (MAMF) Benchmark: mamf-finder.py was derived from research found in The Case for Co-Designing Model Architectures with Hardware paper.

For a detailed discussion and the numbers for various accelerators see Maximum Achievable and Sustainable FLOPS.

While some accelerator manufacturers publish the theoretical TFLOPS these usually can't be reached. As a result of this when we try to optimize our software we have no realistic performance bar to compare ourselves to. The Model FLOPS

Utilization (MFU) metric measures TFLOPS achieved against theoretical TFLOPS. Usually when one scores around 50% MFU it's considered a win. But this gives us no indication how far are we from the real achievable throughput.

This benchmark scans various large shapes of matmul and reports both the highest *achievable* (MAMF — boost burst) and the highest *sustainable* TFLOPS (MSMF — saturated near TDP). As ML workloads are dominated by large matmul operations, MSMF is a realistic bar for sustained training throughput, while MAMF is the short-burst ceiling. Now instead of the previously used MFU, which is not achievable, one can use Model Achievable Matmul FLOPS Utilization (MAMFU) or Model Sustainable Matmul FLOPS Utilization (MSMFU) and actually be able to optimize the workloads to be closer to 100%, in particular with MSFM, since achievable is still not sustainable. Which is very important in order to know when to stop optimizing.

Supported accelerators:

- NVIDIA: Volta, Ampere, Hopper, Blackwell, Rubin
- AMD: MI250, and higher
- Intel Gaudi2/3
- Intel GPUs (A770, A750, B580, etc.)
- Apple silicon (MPS)

`--search auto` derives its shapes from the GPU's compute-unit layout, which `mamf-finder.py` models only for NVIDIA and AMD; on the others give it a range with `--search grid`. Power/clock telemetry, which validates both headlines, exists for NVIDIA, AMD and Gaudi (see the telemetry note below). `mamf-finder-all-gpus.py` runs on NVIDIA and AMD only.

Important notes:

- if you can find a better and more efficient way to detect the best matmul TFLOPS by approaching each new accelerator as a black box, please kindly send a PR with the improvement including the generated log file.
- also if you know that this benchmark should be run under special conditions to show the best results, such as some kernel settings or similar, please submit a PR to add such special instructions. For example, for AMD MI300X I'm being told disabling the numa_balancing is supposed to help.
- since a big part of the overhead comes from HBM IO, if you're using a fused kernel with 2 or more matmuls, whose results don't leave the accelerator's registers, the performance will be definitely faster than what this benchmark reports.
- It also helps to sample your accelerator's actual clock speed. If your accelerator is running at a slower clock than the one used in the spec, there is no chance you can get the theoretical TFLOPS (see How To Calculate Theoretical TFLOPS).

Architecture specific notes:

Follow the special setup instructions before running the benchmark to achieve the best results:

MI300x, MI325X, etc.:

1. Turn numa_balancing off for better performance:

```
sudo sh -c 'echo 0 > /proc/sys/kernel/numa_balancing'
```

2. Enable PyTorch TunableOp:

```
export PYTORCH_TUNABLEOP_ENABLED=1
```

This will make the first iteration very slow, while it's searching for the best GEMM algorithm in the BLAS libraries for each `matmul` shape it encounters, but subsequent operations are likely to be significantly faster than the baseline. See Accelerating models on ROCm using PyTorch TunableOp doc. On ROCm 10, tuning a new shape takes ~2 minutes, which

would turn a ~2,000-shape search into days, so `mamf-finder.py` pauses tuning while it searches (the default kernel is within ~1–2% on good shapes, close enough to rank them) and tunes only the shapes it confirms, before timing them. `--tune tunableop_confirm_max=N` (default 8) caps how many shapes that is; a BF16 `--search auto` run on MI300X then needs about 5 minutes of search plus ~16 minutes of tuning.

On ROCm, `--search auto` prints a note that its wave/tile geometry is only validated on NVIDIA, although on one MI300X it came within 0.1% of a 16,000-shape BF16 grid. A newer stack can move the winning shape rather than the peak: `10240x15360x8192` dropped from 663.6 (ROCm 6.3, torch 2.5.1) to 625.5 (ROCm 10, torch 2.12) while `12288x9728x8192` held ~660 on both, so re-search after a stack upgrade instead of re-timing the old winner. Pick the GPU with `--cuda_device N`, not `HIP_VISIBLE_DEVICES` / `ROCR_VISIBLE_DEVICES`: on some virtualized MI300X hosts, hiding GPU0 left torch with no usable device.

Intel dGPUs (A770, A750, B580, etc.)

- Follow Intel Extension for PyTorch installation steps

AMD / Gaudi telemetry (power / SUSPECT): NVIDIA NVML is the only *validated* backend for excluding unsaturated boost bursts from the headline. `amdsmi` and `pyhlml` are wired and will *report* power/clock when installed, but do not exclude `SUSPECT` shapes until validated on real iron (`VALIDATED_BACKENDS` in `mamf-finder.py`). On NVIDIA, AMD and Gaudi `mamf-finder.py` refuses to start if the telemetry package is missing (`nvidia-ml-py`, `amdsmi`, `habana-pyhlml`) or can't read power/clock; `--telemetry off` runs anyway, with MAMF not boost-validated and no SUSPECT filtering. On MI300X `amdsmi` reads sane Watts/MHz per GPU, but the card hits its 750 W cap within one BF16 kernel and bursts after an idle start *slower*, not faster. So there is no boost burst to filter out, `amdsmi` stays report-only, and AMD MAMF is less reliable than MSMF.

Examples of usage

`K` is the reduction dimension: `(MxK)*(KxN)=(MxN)`. Default dtype is `bfloat16`; `--dtype` also accepts `float16`, `float32`, `float8_e4m3fn` (NVIDIA's fp8), `float8_e4m3fnuz` (AMD MI300's fp8) and `mxfp8`: `float8_e4m3fn` operands with one `float8_e8m0fnu` scale per 32 elements and a `bfloat16` output, which needs hardware MX support such as NVIDIA Blackwell or AMD MI355X. Default iterations are 50 warmup + 100 measured per shape (`--num_warmup_iterations`, `--num_iterations`).

1. Auto search (default) — best the GPU can do anywhere

```
./mamf-finder.py --output_file=$(date +'%Y-%m-%d-%H:%M:%S').txt
# equivalent: ./mamf-finder.py --search auto --output_file=$(date +'%Y-%m-%d-%H:%M:%S').txt
```

Finds near-peak shapes via hardware heuristics and reports **two** headlines. Great for a spec-sheet number; not tied to any particular model. This is what produced the MAMF & MSMF table. On H200, B200 and B300, three fast-search repeats stayed within 0.43% of a 128,000-shape exhaustive scout and beat it on half the GPU/dtype pairs.

- **MAMF** (Maximum *Achievable* Matmul FLOPS) — the boost-clock burst ceiling.
- **MSMF** (Maximum *Sustainable* Matmul FLOPS) — the power-saturated, sustained rate that matches real training throughput. **For picking shapes a real model will use, MSMF is the number that matters.**

On NVIDIA it typically runs in 1–3 minutes including adaptive warmup, the recall screen, thermal soak, and both confirms. On AMD with TunableOp, tuning the confirm shapes adds ~2 minutes per shape, for example it took 20 minutes to run on MI300X (see Architecture specific notes).

How `--search auto` chooses candidate shapes (then both modes share the same confirm phase below):

1. **Wave candidates @ a short K set** — enumerate `(M,N)` shapes that fill an integer number of full waves of thread-block tiles across the SMs (most-square / widest / tallest per wave count 1..16), and measure *every* such `(M,N)` at several Ks (min / mid / max). Covers the high-arithmetic-intensity / wave-perfect basin; measuring all wave candidates (not only a ranked top-N) is what keeps mid/high-K winners in the confirm

set.

2. **Coarse M×N planes @ `Kmin` and K=3072** — cheap grids at the smallest K and the low-K boost basin. The latter is required for B200's non-wave BF16 MAMF family found by exhaustive search.
3. **Tight local grid** around both power-ranked and raw-peak scout seeds (`±4` steps of 256 on M/N, `±2` steps of 1024 on K) — endgame polish for off-axis peaks without letting MSMF-oriented ranking hide a MAMF basin.
4. **MAMF recall screen** — cheaply measure raw scout leaders plus the top several K choices from every wave `(M,N)` layout using the actual idle+burst regime. Promote its leaders into a dedicated MAMF shortlist.

Shared confirm phase (identical for `--search auto` and `--search grid`):

5. **Candidate union + thermal soak** — combine the independently ranked MAMF and MSMF shortlists, then drive the chip to steady state before any sustainable measurement steady state (hot, SM clock settled at the saturated floor), stopping early once the clock stops dropping (`msmf_soak_s`). This is the single biggest reproducibility lever: a cold/cooler card boosts and reads high, so without soaking the MSMF headline would depend on how warm the card happened to be.
6. **MSMF confirm** (sustainable): re-measure every shape in the union with the full iteration count and per-shape warmup, repeated (`confirm_reps`, default 5 → trimmed median), with a short thermal pre-warmup per shape so a freshly switched shape doesn't read cold on its first rep. The confirm set always includes (a) the best scouted K for each most-square low-wave shape (w=1..4) in **both** `(M,N)` orientations — e.g. H200's `1536×2816` family — and (b) the **fattest** scouts (largest `min(M,N,K)`, then volume), which reliably saturate to TDP and anchor the saturated-clock reference. A shape is kept out of the MSMF headline (still reported) if it is **not saturated** — either drawing well below peak power (`SUSPECT`) or running materially **above the saturated-clock floor** (`msmf_clock_ratio`) *while also* drawing below `msmf_sat_power_ratio` of max power. A sparse layout pinned at TDP can hold a higher clock than a fat one; that number is still sustainable and is kept. Jittery shapes (`msmf_max_spread`) are kept out unless nothing else is stable. Finally the winner is **lock-in validated** (`msmf_lock_reps`): re-measured with extra reps and published only if that longer run is itself within the spread tolerance, else the next candidate is tried. All exclusions apply only on validated telemetry backends (currently NVIDIA NVML); untested backends report power/clock but do not exclude.
7. **MAMF confirm** (achievable): re-measure the **same union** as MSMF — not MSMF's candidate list and not a scout-clock pre-filter. A fat shape that saturates while scouting still recovers boost after a short idle (measured on B300). Idle, then a SHORT burst that is **queued and synchronized once**. Synchronizing before each start event (v5–v7) left the GPU idle at the moment the event was taken, which charged kernel-launch latency and the idle DVFS re-ramp to the timed kernel — 1.6–19% depending on shape, and worst for large footprints, so it re-ranked candidates. Per-iteration clocks are instead recovered by projecting each iteration's GPU-time window onto a background sampler's host timeline and taking the minimum clock inside it. The peak is reported only if that clock reached boost (`boost_clock_ratio`). After both headlines are picked, `mamf-finder.py` prints a **same-shape cross-check**: each winning shape measured in both regimes, so the boost→saturated penalty is comparable on the same GEMM.

Each shape is scouted only once, so a candidate that several phases propose costs nothing extra.

Reproducibility — the whole point of a single-run number. A published figure must be reproducible on the same GPU with the same setup, so:

- **Measure while the other GPUs compute.** A real workload runs every GPU of the node at once, and they share the board's power/cooling budget — measured with idle siblings, MSMF gets headroom a full node never has. `mamf-finder-all-gpus.py` runs a continuous matmul on every GPU it is not measuring, which pins it at its power limit, while it measures: first the full search on GPU0, which gives MAMF, MSMF and the MSMF shape, then that shape pinned on each other GPU in turn while all the rest, GPU0 included, run the matmul. The node MSMF is the slowest GPU, since synchronous training runs at its pace; the summary also gives the median and spread across GPUs. GPU0's search gets all your arguments and the pinned runs get them minus the shape selection; the script shows GPU0's console, writes each GPU's log to its own file, and runs everything with the python you start it with, so use the one whose torch you want measured. On 8

H200s the default `--search auto` took about 4.5 minutes and gave a node MSMF of 727 TFLOPS against a median of 752. A single GPU's MSMF moved by up to 5% between runs there, so rerun before calling one GPU slow.

- **Single GPU: run `mamf-finder.py` on its own.** With idle siblings MAMF is a best-case boost burst and MSMF is a single-GPU upper bound. `mamf-finder.py` samples its same-board siblings throughout the MSMF measurement and, if any sat idle, says so under the results with each idle GPU's share of the time.
- **The headline value is the *mean* of the winning shape** (not a lucky max), measured at a known clock, both printed in the headline (… `985W 1360MHz`).
- **To reproduce a published number, re-run its exact shape in grid mode** — e.g. `--m 9472 --n 6144 --k 12288`. Auto's job is to *find* a near-peak shape anywhere; grid *reproduces* (or searches within) a known range and still runs the same MAMF/MSMF confirm. Re-running `auto` re-searches and may land on a different (equivalent) shape.

On a single GPU with idle siblings the MSMF headline is reproducible to ~1–2% run-to-run and MAMF to ~1–2% (boost-locked); most of the residual is genuine saturated-clock jitter. With every other GPU computing at the same time, MSMF varies more between GPUs, as measured below.

What we measured: alone vs all-8 concurrent.

Same protocol on both chips — five `--search auto` runs on GPU0 with siblings idle, then five rounds with the same script on all 8 GPUs at once:

B200 bf16 (torch 2.13.0+cu130):

Setup	MSMF mean	MSMF range	MSMF clock	MAMF mean
GPU0 alone (n=5)	1455.7	1.1%	~1330–1460 MHz	1732.6
GPU0 while all 8 busy (n=5)	1438.7 (-1.2%)	1.8%	~1316–1332 MHz	1757.6
All 8 GPUs × 5 rounds (n=40)	1426.4 (-2.0%)	**10.4%**	1228–1410 MHz	1751.2

H200 bf16 (torch 2.14.0.dev20260810+cu130):

Setup	MSMF mean	MSMF range	MSMF clock	MAMF mean
GPU0 alone (n=5)	702.9	1.4%	~1440–1465 MHz	828.0
GPU0 while all 8 busy (n=5)	701.3 (-0.2%)	2.9%	~1440–1525 MHz	825.8
All 8 GPUs × 5 rounds (n=40)	698.6 (-0.6%)	4.5%	1405–1527 MHz	823.9

Lesson: the other GPUs computing at the same time barely moves **MAMF** (boost bursts still hit the boost clock — they draw only ~150–300 W, so board power/cooling still has headroom). It *does* move **MSMF**, and the severity is board-dependent: on B200 (1000 W TDP) concurrent siblings pulled GPU0's saturated clock down ~50 MHz (-1.2% MSMF) and opened a **~10% MSMF spread across the 8 GPUs**; on H200 (700 W TDP) the same protocol was much milder (-0.2% on GPU0, ~4.5% across the board). So the alone-GPU MSMF overstates what a full node sustains: a full-node training run sees the concurrent distribution, and since it waits on its slowest GPU, the bottom of that range is what it actually gets. How far that sits below the alone number depends on the board's shared power/cooling budget. In the results table, `Sib = yes` is the `mamf-finder-all-gpus.py` measurement, every other GPU running a continuous matmul (MSMF is the slowest GPU); `Sib = no` is still one GPU with siblings idle, a single-GPU upper bound.

`mamf-finder.py -h` lists the options. The algorithm's own knobs, the names in backticks in the steps above (`confirm_reps`, `msmf_soak_s`, …), live in the `Tuning` class near the end of the script with their defaults and a line on what each does. The

published numbers were measured with those defaults; to try another value pass e.g. `--tune msmf_soak_s=30 --tune confirm_reps=7`.

Reading the output

The console shows each phase as a title, a column header and one row that is updated in place as shapes are measured; when the phase ends, its winning row stays on screen:

```
Grid search: sweeping 4x1x1 shapes (scout 20 iters / 8 warmup), then confirm ...
     #  MxNxK                mean median    max   best     W   MHz
     4  4096x4096x4096      779.5  779.3  784.2  779.5   590  1980  <- best mean

MAMF recall screen: 4 shapes ...
     #  MxNxK                peak   MHz
     1  4096x4096x4096      777.7  1980  <- best peak

MSMF (sustainable) confirm: 4 shapes ...
  MxNxK              TFLOPS     W   MHz spread  runs
  4096x4096x4096      743.0   679  1792   1.6%  749.6 743.2 740.4 738.7 740.3 737.6 743.0 743.3
746.2  <- MSMF

MAMF (achievable) confirm: 4 shapes ...
  MxNxK              TFLOPS     W   MHz  top peaks
  4096x4096x4096      780.5   332  1980  780.5 780.1 779.3 779.1 778.2  <- MAMF

--------------------------------------------------------------------------------

** Results:

Tried 4 shapes => the best outcomes were:
MAMF (max achievable,  boost burst): 781 TFLOPS @ 4096x4096x4096 (MxNxK)  332W 1980MHz
MSMF (max sustainable, saturated):   743 TFLOPS @ 4096x4096x4096 (MxNxK)  679W 1792MHz
```

The MSMF row shows the lock-in re-measurement (9 runs), which is what the headline uses.

`W` and `MHz` are live NVML (or amdsmi/hlml) samples taken during the timed loop. On a short MAMF burst NVML's power sample can lag the kernel: on A100 PCIe a ~80 ms burst after an idle reads near-idle watts, so that `W` is not the burst's draw. The MAMF and MSMF headlines are integer TFLOPS; a half rounds up. Every row, plus the details behind each decision (confirm candidates, boost reference, shapes excluded from MSMF and why, lock-in, same-shape cross-check), is written to the `--output_file` log. Shapes that ran well below the run's peak power are excluded from the **MSMF** (sustainable) headline on validated backends — a low-power boost burst is not sustainable — while boost readings feed the **MAMF** (achievable) headline (see MAMF & MSMF).

What makes a fast shape

Peak GEMM shapes are not mysterious; they follow the paper's recipe (arXiv:2401.14489):

1. **Tensor-core alignment** — M, N, K multiples of `128 B / sizeof(dtype)` elements (bf16→64, fp8→128).
2. **Tile quantization** — output tiles cleanly into the kernel's tile (canonical efficient tile is 128×256), so M,N multiples of 256 are safe.
3. **Wave quantization** — thread-block count ≈ an exact multiple of the SM count, so there is no partial tail wave. Winners typically have `wave_eff ≥ ~0.99`.

4. **K only sets arithmetic intensity** — it does not enter the tile/wave math. Larger K → more compute-bound. Empirically there are *two* high-TFLOPS basins: high-AI (large K) and many-waves (min-K, huge M×N) — which is why `auto` covers both.

For model-design work (picking shapes your layers will emit), also see Vector and matrix size divisibility.

2. Grid search — best shape in *your* range (the practical case)

Use `--search grid` (or just pass any `--m/--n/--k/_*_range` argument — that implies grid) when you care about a **specific subspace**: the shapes your new model will actually emit, a single training shape, or an accelerator-specific band you want to map exhaustively.

Grid reports the **same MAMF + MSMF headlines** as auto. The only difference is how candidates are chosen: auto derives them from hardware heuristics; grid sweeps the M/N/K range *you* give, then runs the shared confirm phase. For model work, **MSMF in your range** is the useful answer — not the GPU's absolute peak.

```
# shapes your model will use (example: M in 2k..8k, N=K=4096)
./mamf-finder.py --m_range 2048 8193 256 --n 4096 --k 4096 --output_file=$(date
+'%Y-%m-%d-%H:%M:%S').txt

# one exact shape (reproduce a published headline)
./mamf-finder.py --m 1024 --n 1024 --k 1024 --output_file=$(date +'%Y-%m-%d-%H:%M:%S').txt

# fp8 sweep over a coarse lattice
./mamf-finder.py --m_range 0 20480 1024 --n_range 0 20480 1024 --k_range 0 20480 1024 \
    --dtype float8_e4m3fn --output_file=$(date +'%Y-%m-%d-%H:%M:%S').txt
```

Give the range enough room to saturate. MSMF needs at least one fat/high-K shape that can pin the saturated-clock floor. A narrow/small grid (e.g. only `K=4096` squares) can leave every candidate still boosting or jittery; `mamf-finder.py` then warns and falls back to a "best available" MSMF. Widen M/N/K (especially `min(M,N,K)` and K) so a truly sustainable shape is in the set.

You can Ctrl-C a long grid run and still get the best result so far. Finer steps (512 / 256 instead of 1024) cost 8× / 64× wall time. For which shapes tend to peak on a given accelerator, see Vector and matrix size divisibility.

Architecture-specific setup (MI300X `numa_balancing` / TunableOp, Intel dGPU install) is under Architecture specific notes above.

Results

The measurements that I have gathered so far can be found at Maximum Achievable and Sustainable Matmul FLOPS comparison table. When I had access to a particular accelerator I run the benchmarks myself, when I didn't it was the kind contributors who invested their time to get these numbers. So I'm very grateful to those.

Troubleshooting NVIDIA GPUs

Glossary

- DBE: Double Bit ECC Error
- DCGM: (NVIDIA) Data Center GPU Manager
- ECC: Error-Correcting Code
- FB: Frame Buffer
- SBE: Single Bit ECC Error
- SDC: Silent Data Corruption

Errors and diagnostics

See also Diagnosing crashes, hangs and tracing execution for diagnosing crashes from the software side.

Xid Errors

No hardware is perfect, sometimes due to the manufacturing problems or due to tear and wear (especially because of exposure to high heat), GPUs are likely to encounter various hardware issues. A lot of these issues get corrected automatically without needing to really understand what's going on. If the application continues running usually there is nothing to worry about. If the application crashes due to a hardware issue it's important to understand why this is so and how to act on it.

A normal user who uses a handful of GPUs is likely to never need to understand GPU-related hardware issues, but if you come anywhere close to massive ML training where you are likely to use hundreds to thousands of GPUs it's certain that you'd want to understand about different hardware issues.

In your system logs you are likely to see occasionally Xid Errors like:

```
NVRM: Xid (PCI:0000:10:1c): 63, pid=1896, Row Remapper: New row marked for remapping, reset gpu
to activate.
```

To get those logs one of the following ways should work:

```
sudo grep Xid /var/log/syslog
sudo dmesg -T | grep Xid
```

Typically, as long as the training doesn't crash, these errors often indicate issues that automatically get corrected by the hardware.

The full list of Xid Errors and their interpretation can be found here.

You can run `nvidia-smi -q` and see if there are any error counts reported. For example, in this case of Xid 63, you will see something like:

```
Timestamp                                 : Wed Jun  7 19:32:16 2023
Driver Version                            : 510.73.08
```

```
CUDA Version                            : 11.6

Attached GPUs                           : 8
GPU 00000000:10:1C.0
    Product Name                        : NVIDIA A100-SXM4-80GB
    [...]
    ECC Errors
        Volatile
            SRAM Correctable            : 0
            SRAM Uncorrectable          : 0
            DRAM Correctable            : 177
            DRAM Uncorrectable          : 0
        Aggregate
            SRAM Correctable            : 0
            SRAM Uncorrectable          : 0
            DRAM Correctable            : 177
            DRAM Uncorrectable          : 0
    Retired Pages
        Single Bit ECC                  : N/A
        Double Bit ECC                  : N/A
        Pending Page Blacklist          : N/A
    Remapped Rows
        Correctable Error               : 1
        Uncorrectable Error             : 0
        Pending                         : Yes
        Remapping Failure Occurred      : No
        Bank Remap Availability Histogram
            Max                         : 639 bank(s)
            High                        : 1 bank(s)
            Partial                     : 0 bank(s)
            Low                         : 0 bank(s)
            None                        : 0 bank(s)
[...]
```

Here we can see that Xid 63 corresponds to:

```
ECC page retirement or row remapping recording event
```

which may have 3 causes: HW Error / Driver Error / FrameBuffer (FB) Corruption

This error means that one of the memory rows is malfunctioning and that upon either reboot and/or a gpu reset one of the 640 spare memory rows (in A100) will be used to replace the bad row. Therefore we see in the report above that only 639 banks remain (out of 640).

The Volatile section of the `ECC Errors` report above refers to the errors recorded since last reboot/GPU reset. The Aggregate section records the same error since the GPU was first used.

Now, there are 2 types of errors - Correctable and Uncorrectable. The correctable one is a Single Bit ECC Error (SBE) where despite memory being faulty the driver can still recover the correct value. The uncorrectable one is where more than one bit is faulty and it's called Double Bit ECC Error (DBE). Typically, the driver will retire whole memory pages if 1 DBE or 2 SBE errors occur at the same memory address. For full information see this document

A correctable error will not impact the application, a non-correctable one will crash the application. The memory page containing the uncorrectable ECC error will be blacklisted and not accessible until the GPU is reset.

If there are page scheduled to be retired you will see something like this in the output of `nvidia-smi -q`:

```
    Retired pages
        Single Bit ECC             : 2
        Double Bit ECC             : 0
        Pending Page Blacklist    : Yes
```

Each retired page decreases the total memory available to applications. But the maximum amount of pages retired amounts to only 4MiB in total, so it doesn't reduce the total available GPU memory by much.

To dive even deeper into the GPU debugging, please refer to this document - it includes a useful triage chart which helps to determine when to RMA GPUs. This document has additional information about Xid 63-like errors

For example it suggests:

> If associated with XID 94, the application that encountered the error needs to be restarted. All other applications on the system can keep running as is until there is a convenient time to reboot for row remapping to activate.

If after a reboot the same condition occurs for the same memory address, memory remapping has failed and Xid 64 is emitted. If it keeps happening after reboot, the GPU needs to be RMA'ed.

When that happens it shows up in the same `Remapped Rows` block as above, with the failure flag set and a bank that has run out of spares. This is Crusoe Cloud's published signature for a GPU that qualifies for replacement:

```
Remapped Rows
        Correctable Error                 : 0
        Uncorrectable Error               : 0
        Pending                           : No
        Remapping Failure Occurred        : Yes
        Bank Remap Availability Histogram
            Max                           : 639 bank(s)
            High                          : 0 bank(s)
            Partial                       : 0 bank(s)
            Low                           : 0 bank(s)
            None                          : 1 bank(s)
```

Note that not a single row was remapped successfully, yet a failure has been recorded, and one bank sits at `None` - it has no reserved rows left to remap into. That combination is what distinguishes a GPU that needs replacing from one that just needs a reset. To read only this block rather than the whole report, use `nvidia-smi -q -d ROW_REMAPPER`.

At other times you may get Xid 63 or 64 and the application will crash, which usually generates additional Xid errors, but most of the time it means that the error was uncorrectable (i.e. it was a DBE sort of an error and then it'll be Xid 48).

As mentioned earlier to reset a GPU you can either simply reboot the machine, or run:

```
nvidia-smi -r -i gpu_id
```

where `gpu_id` is the sequential number of the gpu you want to reset, e.g. `0` for the first GPU. Without `-i` all GPUs will be reset.

uncorrectable ECC error encountered

If you get an error:

```
CUDA error: uncorrectable ECC error encountered
```

as in the previous section, checking the output of `nvidia-smi -q` this time for `ECC Errors` entries will tell which GPU is the problematic one. But if you need to do a quick check in order to recycle a node if it has at least one GPU with this issue, you can just do this:

```
$ nvidia-smi -q | grep -i correctable | grep -v 0
            SRAM Uncorrectable            : 1
            SRAM Uncorrectable            : 5
```

On a good node, this should return nothing, as all counters should be `0`. But in the example above we had one broken GPU - there were two entries because the full record was:

```
    ECC Errors
        Volatile
            SRAM Correctable              : 0
            SRAM Uncorrectable            : 1
            DRAM Correctable              : 0
            DRAM Uncorrectable            : 0
        Aggregate
            SRAM Correctable              : 0
            SRAM Uncorrectable            : 5
            DRAM Correctable              : 0
            DRAM Uncorrectable            : 0
```

The first entry is for `Volatile` (errors counted since the last time the GPU driver reload) and the second is for `Aggregate` (total errors counter for the whole life time of the GPU). In this example we see a Volatile counter for SRAM Uncorrectable errors to be 1 and for the life-time counter it's 5 - that is this is not the first time the GPU runs into this problem.

This typically would correspond to Xid 94 error (see: Xid Errors, most likely w/o Xid 48).

To overcome this issue as in the previous section, reset the problematic GPU:

```
nvidia-smi -r -i gpu_id
```

Rebooting the machine will have the same effect.

Now when it comes to Aggregate SRAM Uncorrectable errors, if you have more than 4, that's usually a reason to RMA that GPU.

Running diagnostics

If you suspect one or more NVIDIA GPUs are broken on a given node, `dcgmi` is a great tool to quickly find any bad GPUs.

NVIDIA® Data Center GPU Manager (DCGM) is documented here and can be downloaded from here.

Here is an example slurm script that will run very in-depth diagnostics (`-r 3`), which will take about 10 minutes to complete on an 8-GPU node:

```
$ cat dcgmi-1n.slurm
#!/bin/bash
#SBATCH --job-name=dcgmi-1n
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=96
#SBATCH --gres=gpu:8
#SBATCH --exclusive
#SBATCH --output=%x-%j.out

set -x -e
echo "START TIME: $(date)"
srun --cpus-per-task=$SLURM_CPUS_PER_TASK --output=%x-%j-%N.out dcgmi diag -r 3
echo "END TIME: $(date)"
```

Now to run it on specific nodes of choice:

```
sbatch --nodelist=node-115 dcgmi-1n.slurm
sbatch --nodelist=node-151 dcgmi-1n.slurm
sbatch --nodelist=node-170 dcgmi-1n.slurm
```

edit the nodelist argument to point to the node name to run.

If the node is drained or downed and you can't launch a slurm job using this node, just `ssh` into the node and run the command directly on the node:

```
dcgmi diag -r 3
```

If the diagnostics didn't find any issue, but the application still fails to work, re-run the diagnostics with level 4, which will now take more than 1 hour to complete:

```
dcgmi diag -r 4
```

footnote: apparently silent data corruptions (SDC) can only be detected with `dcgmi diag -r 4` and even then some might be missed. This problem happens occasionally and you may not even be aware that your GPU is messing up the `matmul` at times. I'm pretty sure we had this happen to us, as we were getting weird glitches during training and I spent many days with the NVIDIA team diagnosing the problem, but we failed to do so - eventually the problem disappeared probably because the bad GPU(s) got replaced due to reported failures.

For example, if you run into a repeating Xid 64 error it's likely that the diagnostics report will include:

```
+---------------------------+-------------------------------------------------+
| Diagnostic                | Result                                          |
+===========================+=================================================+
|-----  Deployment  --------+-------------------------------------------------|
| Error                     | GPU 3 has uncorrectable memory errors and row   |
|                           |  remappings are pending                         |
```

so you now know to RMA that problematic GPU, if remapping fails.

But, actually, I found that most of the time `-r 2` already detects faulty GPUs. And it takes just a few minutes to complete. Here is an example of the `-r 2` output on a faulty node:

```
| GPU Memory                | Pass - GPUs: 1, 2, 3, 4, 5, 6, 7                |
|                           | Fail - GPU: 0                                   |
| Warning                   | GPU 0 Thermal violations totaling 13.3 second   |
|                           | s started at 9.7 seconds into the test for GP   |
|                           | U 0 Verify that the cooling on this machine i   |
|                           | s functional, including external, thermal mat   |
|                           | erial interface, fans, and any other componen   |
|                           | ts.
```

The `dcgmi` tool contains various other levels of diagnostics, some of which complete in a matter of a few minutes and can be run as a quick diagnostic in the epilogue of SLURM jobs to ensure that the node is ready to work for the next SLURM job, rather than discovering that after the user started their job and it crashed.

When filing an RMA report you will be asked to run `nvidia-bug-report.sh` script, the output of which you will need to submit with the RMA request.

I usually save the log as well for posterity using one of:

```
dcgmi diag -r 2 | tee -a dcgmi-r2-`hostname`.txt
dcgmi diag -r 3 | tee -a dcgmi-r3-`hostname`.txt
dcgmi diag -r 4 | tee -a dcgmi-r4-`hostname`.txt
```

Health and inventory checks

How to get the VBIOS info

GPU VBIOS version might be important when researching issues. Let's add the name and bus id to the query, we get:

```
$ nvidia-smi --query-gpu=gpu_name,gpu_bus_id,vbios_version --format=csv
name, pci.bus_id, vbios_version
NVIDIA H100 80GB HBM3, 00000000:04:00.0, 96.00.89.00.01
[...]
NVIDIA H100 80GB HBM3, 00000000:8B:00.0, 96.00.89.00.01
```

Hint: to query for dozens of other things, run:

```
nvidia-smi --help-query-gpu
```

How to check if your GPU's PCIe generation is supported

Check the PCIe bandwidth reports from the system's boot messages:

```
$ sudo dmesg | grep -i 'limited by'
[   10.735323] pci 0000:04:00.0: 252.048 Gb/s available PCIe bandwidth, limited by 16.0 GT/s PCIe
x16 link at 0000:01:00.0 (capable of 504.112 Gb/s with 32.0 GT/s PCIe x16 link)
[...]
[   13.301989] pci 0000:8b:00.0: 252.048 Gb/s available PCIe bandwidth, limited by 16.0 GT/s PCIe
x16 link at 0000:87:00.0 (capable of 504.112 Gb/s with 32.0 GT/s PCIe x16 link)
```

In this example, as PCIe 5 spec is 504Gbps, you can see that on this node only half of the possible bandwidth is usable, because the PCIe switch is gen4. For PCIe specs see this.

Since most likely you have NVLink connecting the GPUs to each other, this shouldn't matter for GPU to GPU comms, but it'd slow down any data movement between the GPU and the host, as the data speed is limited by the speed of the slowest link.

How to check error counters of NVLink links

If you're concerned your NVLink malfunctions you can check its error counters:

```
$ nvidia-smi nvlink -e
GPU 0: NVIDIA H100 80GB HBM3 (UUID: GPU-abcdefab-cdef-abdc-abcd-abababababab)
         Link 0: Replay Errors: 0
         Link 0: Recovery Errors: 0
         Link 0: CRC Errors: 0

         Link 1: Replay Errors: 0
         Link 1: Recovery Errors: 0
         Link 1: CRC Errors: 0

         [...]

         Link 17: Replay Errors: 0
         Link 17: Recovery Errors: 0
         Link 17: CRC Errors: 0
```

Another useful command is:

```
$ nvidia-smi nvlink --status
GPU 0: NVIDIA H100 80GB HBM3 (UUID: GPU-abcdefab-cdef-abdc-abcd-abababababab)
         Link 0: 26.562 GB/s
```

```
        [...]
        Link 17: 26.562 GB/s
```

this one tells you the current speed of each link

Run `nvidia-smi nvlink -h` to discover more features (reporting, resetting counters, etc.).

How to check GPU memory row-remapping health

Row remapping is how Ampere and later GPUs deal with memory that has gone bad - a degrading bank row is replaced by one of the spares that every HBM bank reserves for the purpose. It is NVIDIA's replacement for the page retirement scheme used by earlier generations, with a much larger budget: up to 512 remappings for the frame buffer, against 64 retirements before it. Xid Errors covers what this looks like once a GPU has thrown an Xid 63 or 64 - this section is about checking it deliberately, before a job dies.

That section reads the state out of `nvidia-smi -q`, which prints a long block per GPU. For checking a whole node there is a CSV query that gives one line per GPU instead:

```
$ nvidia-smi
--query-remapped-rows=gpu_name,gpu_bus_id,remapped_rows.failure,remapped_rows.pending,remapped_rows.correctable,r
--format=csv
gpu_name, gpu_bus_id, remapped_rows.failure, remapped_rows.pending, remapped_rows.correctable,
remapped_rows.uncorrectable
NVIDIA H200, 00000000:59:00.0, No, No, 0, 0
[...]
NVIDIA H200, 00000000:A5:00.0, No, No, 0, 0
```

That is a healthy node - no failures, nothing pending, no rows remapped. To see the other end of it, Xid Errors shows the same fields from a GPU whose row remapping has failed. Reading a report that isn't all `No` and `0`:

- `correctable` and `uncorrectable` count the rows that have already been remapped, after repeated SBEs and after a DBE respectively. A non-zero count is not by itself a reason to pull the GPU out of service - it means the sparing did its job.
- `pending: Yes` means a remap has been decided but doesn't take effect until the GPU is reset, and it does not put the running job at risk of touching the bad cell. It does change how much memory that GPU has, though: when the trigger was an uncorrectable error the driver offlines the page containing it immediately, and that page stays out of the allocatable pool until the reset remaps the row in hardware and hands the address space back - so a job sized to fill HBM can start hitting OOM on this GPU while still fitting on a healthy one.
- `failure: Yes` means a remap was attempted and did not succeed. This is the one that means RMA rather than reset-and-return.

The same fields are available from `--query-gpu` if you want them alongside other per-GPU columns, e.g. `nvidia-smi --query-gpu=gpu_bus_id,remapped_rows.pending,remapped_rows.failure --format=csv`.

Hint: for the full field list, including the `remapped_rows.sbe/.dbe` aliases, run:

```
nvidia-smi --help-query-remapped-rows
```

How to detect if a node is missing GPUs

If you got a new VM, there are odd cases where there is less than expected number of GPUs. Here is how you can quickly test you have got 8 of them:

```
cat << 'EOT' >> test-gpu-count.sh
#!/bin/bash

set -e

# test the node has 8 gpus
test $(nvidia-smi -q | grep UUID | wc -l) != 8 && echo "broken node: not exactly 8 gpus" && false
EOT
```

and then:

```
bash test-gpu-count.sh
```

How to detect if you get the same broken node again and again

This is mostly relevant to cloud users who rent GPU nodes.

So you launched a new virtual machine and discovered it has one or more broken NVIDIA GPUs. You discarded it and launched a new and the GPUs are broken again.

Chances are that you're getting the same node with the same broken GPUs. Here is how you can know that.

Before discarding the current node, run and log:

```
$ nvidia-smi -q | grep UUID
    GPU UUID                              : GPU-2b416d09-4537-ecc1-54fd-c6c83a764be9
    GPU UUID                              : GPU-0309d0d1-8620-43a3-83d2-95074e75ec9e
    GPU UUID                              : GPU-4fa60d47-b408-6119-cf63-a1f12c6f7673
    GPU UUID                              : GPU-fc069a82-26d4-4b9b-d826-018bc040c5a2
    GPU UUID                              : GPU-187e8e75-34d1-f8c7-1708-4feb35482ae0
    GPU UUID                              : GPU-43bfd251-aad8-6e5e-ee31-308e4292bef3
    GPU UUID                              : GPU-213fa750-652a-6cf6-5295-26b38cb139fb
    GPU UUID                              : GPU-52c408aa-3982-baa3-f83d-27d047dd7653
```

These UUIDs are unique to each GPU.

When you then re-created your VM, run this command again - if the UUIDs are the same - you know you have the same broken GPUs.

To automate this process so that you always have this data as it'd be too late if you already rebooted the VM, add somewhere in your startup process this:

```
nvidia-smi -q | grep UUID > nvidia-uuids.$(hostname).$(date '+%Y-%m-%d-%H:%M').txt
```

You'd want to save the log file on some persistent filesystem for it to survive reboot. If you do not have one make it local and immediately copy to the cloud. That way it'll always be there when you need it.

Sometimes just rebooting the node will get new hardware. In some situations you get new hardware on almost every reboot, in other situations this doesn't happen. And this behavior may change from one provider to another.

If you keep on getting the same broken node - one trick to overcoming this is allocating a new VM, while holding the broken VM running and when the new VM is running - discarding the broken one. That way you will surely get new GPUs - except there is no guarantee they won't be broken as well. If the use case fits consider getting a static cluster where it's much easier to keep the good hardware.

This method is extra-crucial for when GPUs don't fail right away but after some use so it is non-trivial to see that there is a problem. Even if you reported this node to the cloud provider the technician may not notice the problem right away and put the bad node back into circulation. So if you're not using a static cluster and tend to get random VMs on demand you may want to keep a log of bad UUIDs and know you have got a lemon immediately and not 10 hours into the node's use.

Cloud providers usually have a mechanism of reporting bad nodes. Therefore other than discarding a bad node, it'd help yourself and other users to report bad nodes. Since most of the time users just discard the bad nodes, the next user is going to get them. I have seen users getting a very high percentage of bad nodes in some situations.

Metrics

How to get the real GPU utilization metrics

As explained here the `GPU-Util` column in the `nvidia-smi` output isn't really telling you the GPU Utilization. What it's telling you is the percentage of time during which one or more kernels were executing on the GPU. It's not telling you whether a single SM is being used or all of them. So even if you run a tiny `matmul` all the time, you may get a very high gpu util, while most of the GPU isn't doing anything.

footnote: I have seen GPU util column showing 100% on all gpus when one GPU would stop responding and then whole machinery was blocked waiting for that gpu to respond. Which is how I discovered that it couldn't be showing the real GPU utilization in the first place.

What you want to measure instead is GPU's utilization of the available capacity, otherwise known as "saturation". Alas, this information isn't provided by `nvidia-smi`. In order to get this information you need to install dcgm-exporter (which in turn currently requires a recent golang and DCGM (`datacenter-gpu-manager`) and a root access).

Please note that this tool works only high-end data center NVIDIA GPUs, so if you have a consumer level GPU it won't work.

After installing the prerequisites I built the tool:

```
git clone https://github.com/NVIDIA/dcgm-exporter.git
cd dcgm-exporter
make binary
```

And then I was able to get the "real" utilization metrics described in the article with this `dcgm-exporter` config file:

```
$ cat << EOT > dcp-metrics-custom.csv
DCGM_FI_PROF_SM_OCCUPANCY,       gauge, The ratio of number of warps resident on an SM.
DCGM_FI_PROF_PIPE_TENSOR_ACTIVE, gauge, Ratio of cycles the tensor (HMMA) pipe is active.
DCGM_FI_PROF_PIPE_FP16_ACTIVE,   gauge, Ratio of cycles the fp16 pipes are active.
DCGM_FI_PROF_PIPE_FP32_ACTIVE,   gauge, Ratio of cycles the fp32 pipes are active.
EOT
```

Then I launched the daemon (root is required):

```
$ sudo cmd/dcgm-exporter/dcgm-exporter -c 500 -f dcp-metrics-custom.csv
[...]
INFO[0000] Starting webserver
INFO[0000] Listening on                                  address="[::]:9400"
```

`-c 500` refreshes every 0.5sec

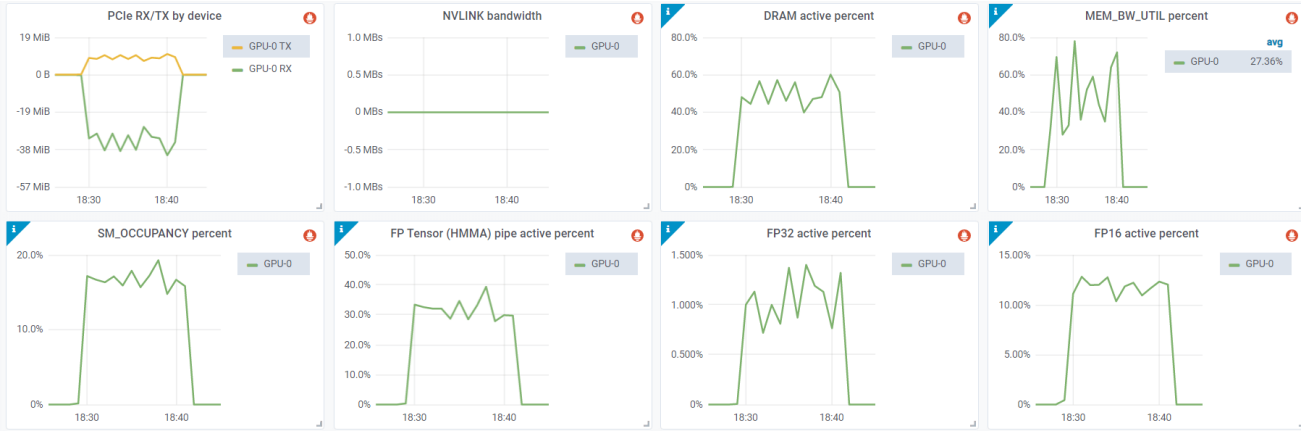
and now I was able poll it via:

```
watch -n 0.5 "curl http://localhost:9400/metrics"
```

by running it in one console, and launching a GPU workload in another console. The last column of the output is the utilization of these metrics (where `1.0 == 100%`).

`etc/dcp-metrics-included.csv` from the repo contains all the available metrics, so you can add more metrics.

This is a quick way of doing that, but the intention is to use it with Prometheus which will give you nice charts. E.g. the article included an example where you can see the SM occupancy, Tensor core, FP16 and FP32 Core utilization in the second row of the charts:



(source)

For completion here is an example from the same article showing a 100% gpu util with a CUDA kernel that is doing absolutely nothing compute-wise other than occupying a single Streaming Multiprocessor (SM):

```
$ cat << EOT > 1_sm_kernel.cu
__global__ void simple_kernel() {
    while (true) {}
}

int main() {
    simple_kernel<<<1, 1>>>();
    cudaDeviceSynchronize();
}
EOT
```

Let's compile it:

```
nvcc 1_sm_kernel.cu -o 1_sm_kernel
```

And now run it in console A:

```
$ ./1_sm_kernel
```

and in console B:

```
$ nvidia-smi
Tue Oct  8 09:49:34 2024
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.90.12              Driver Version: 550.90.12      CUDA Version: 12.4     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA A100 80GB PCIe          Off |   00000000:01:00.0 Off |                    0 |
| N/A   32C    P0             69W /  300W |     437MiB /  81920MiB |    100%      Default |
|                                         |                        |             Disabled |
```

You can see the `100%` GPU-Util. So here 1 SM is used whereas A100-80GB PCIe has 108 SMs! And it's not even doing any compute as it just runs an infinite loop of doing nothing.

Troubleshooting AMD GPUs

XXX: this is very early - collecting various tools/notes

As most of us are well familiar with NVIDIA tools, I will try to provide the mapping where possible to the familiar tools.

Tools

HIP_VISIBLE_DEVICES / CUDA_VISIBLE_DEVICES / ROCR_VISIBLE_DEVICES

ROCm has more than one "visible devices" knob because isolation can happen at different layers of the stack. For PyTorch on ROCm the usual CUDA-style choice is HIP-level:

```
HIP_VISIBLE_DEVICES=0,1 python my-program.py
```

`CUDA_VISIBLE_DEVICES` is honored the same way (CUDA-compat alias for HIP on AMD).

That only filters what the **HIP** runtime exposes. ROCr can still initialize every GPU, and non-HIP clients sitting on ROCr (OpenCL, AOMP, UCX, ...) are unaffected.

To hide devices from the whole user-mode ROCm stack, filter at **ROCr**:

```
ROCR_VISIBLE_DEVICES=0,1 python my-program.py
```

`ROCR_VISIBLE_DEVICES` also accepts UUID strings (HIP's list is indices only). If both are set, HIP sees only the devices ROCr left visible, so HIP indices are relative to that filtered set.

On Linux, AMD's isolation docs lean on `ROCR_VISIBLE_DEVICES` when you want stack-wide isolation; for ordinary single-framework PyTorch work, `HIP_VISIBLE_DEVICES` / `CUDA_VISIBLE_DEVICES` is enough. See GPU isolation techniques.

rocm-smi

`rocm-smi` (`nvidia-smi` equivalent) shows a condensed state of all the ROCm accelerators.

For example here is an 8xMI300X node:

```
$ rocm-smi
======================================== ROCm System Management Interface
========================================
================================================== Concise Info
==================================================
Device  [Model : Revision]    Temp        Power     Partitions      SCLK    MCLK    Fan  Perf
PwrCap  VRAM%  GPU%
        Name (20 chars)       (Junction)  (Socket)  (Mem, Compute)
================================================================================================================
0       [0x74a1 : 0x00]       45.0°C      173.0W    NPS1, SPX       132Mhz  900Mhz  0%   auto
750.0W    0%   0%
```

```
        AMD Instinct MI300X
1       [0x74a1 : 0x00]       41.0°C      179.0W    NPS1, SPX       132Mhz  900Mhz  0%   auto
750.0W    0%   0%
        AMD Instinct MI300X
2       [0x74a1 : 0x00]       47.0°C      180.0W    NPS1, SPX       131Mhz  900Mhz  0%   auto
750.0W    0%   0%
        AMD Instinct MI300X
3       [0x74a1 : 0x00]       45.0°C      178.0W    NPS1, SPX       131Mhz  900Mhz  0%   auto
750.0W   17%   0%
        AMD Instinct MI300X
4       [0x74a1 : 0x00]       45.0°C      175.0W    NPS1, SPX       132Mhz  900Mhz  0%   auto
750.0W    0%   0%
        AMD Instinct MI300X
5       [0x74a1 : 0x00]       43.0°C      175.0W    NPS1, SPX       132Mhz  900Mhz  0%   auto
750.0W    0%   0%
        AMD Instinct MI300X
6       [0x74a1 : 0x00]       45.0°C      175.0W    NPS1, SPX       132Mhz  900Mhz  0%   auto
750.0W    0%   0%
        AMD Instinct MI300X
7       [0x74a1 : 0x00]       43.0°C      176.0W    NPS1, SPX       132Mhz  900Mhz  0%   auto
750.0W    0%   0%
        AMD Instinct MI300X
================================================================================================================================
============================================= End of ROCm SMI Log
==============================================
```

Oddly it shows no real memory usage - only the percentage, which isn't very practical.

A handy alias to watch updates in real time:

```
alias wr='watch -n 1 rocm-smi'
```

rocminfo

`rocminfo` (`nvidia-smi -q` equivalent) shows the detailed information about each accelerator.

This one shows both the CPU and the GPU information

Here is a snippet for cpu0 and gpu0 (note it starts counting the cpus as nodes 0..1, and then GPUs as nodes 2..9):

```
$ rocminfo
ROCk module is loaded
=====================
HSA System Attributes
=====================
Runtime Version:         1.1
System Timestamp Freq.:  1000.000000MHz
Sig. Max Wait Duration:  18446744073709551615 (0xFFFFFFFFFFFFFFFF) (timestamp count)
Machine Model:           LARGE
System Endianness:       LITTLE
```

```
Mwaitx:                  DISABLED
DMAbuf Support:          YES

==========
HSA Agents
==========
*******
Agent 1
*******
  Name:                    AMD EPYC 9534 64-Core Processor
  Uuid:                    CPU-XX
  Marketing Name:          AMD EPYC 9534 64-Core Processor
  Vendor Name:             CPU
  Feature:                 None specified
  Profile:                 FULL_PROFILE
  Float Round Mode:        NEAR
  Max Queue Number:        0(0x0)
  Queue Min Size:          0(0x0)
  Queue Max Size:          0(0x0)
  Queue Type:              MULTI
  Node:                    0
  Device Type:             CPU
  Cache Info:
    L1:                      32768(0x8000) KB
  Chip ID:                 0(0x0)
  ASIC Revision:           0(0x0)
  Cacheline Size:          64(0x40)
  Max Clock Freq. (MHz):   2450
  BDFID:                   0
  Internal Node ID:        0
  Compute Unit:            128
  SIMDs per CU:            0
  Shader Engines:          0
  Shader Arrs. per Eng.:   0
  WatchPts on Addr. Ranges:1
  Features:                None
  Pool Info:
    Pool 1
      Segment:                 GLOBAL; FLAGS: FINE GRAINED
      Size:                    792303268(0x2f3996a4) KB
      Allocatable:             TRUE
      Alloc Granule:           4KB
      Alloc Alignment:         4KB
      Accessible by all:       TRUE
    Pool 2
      Segment:                 GLOBAL; FLAGS: KERNARG, FINE GRAINED
      Size:                    792303268(0x2f3996a4) KB
      Allocatable:             TRUE
      Alloc Granule:           4KB
      Alloc Alignment:         4KB
      Accessible by all:       TRUE
```

```
    Pool 3
      Segment:                 GLOBAL; FLAGS: COARSE GRAINED
      Size:                    792303268(0x2f3996a4) KB
      Allocatable:             TRUE
      Alloc Granule:           4KB
      Alloc Alignment:         4KB
      Accessible by all:       TRUE
  ISA Info:
[...]

  Name:                    gfx942
  Uuid:                    GPU-ababaeeffecddc50
  Marketing Name:          AMD Instinct MI300X
  Vendor Name:             AMD
  Feature:                 KERNEL_DISPATCH
  Profile:                 BASE_PROFILE
  Float Round Mode:        NEAR
  Max Queue Number:        128(0x80)
  Queue Min Size:          64(0x40)
  Queue Max Size:          131072(0x20000)
  Queue Type:              MULTI
  Node:                    2
  Device Type:             GPU
  Cache Info:
    L1:                      16(0x10) KB
    L2:                      8192(0x2000) KB
  Chip ID:                 29857(0x74a1)
  ASIC Revision:           1(0x1)
  Cacheline Size:          64(0x40)
  Max Clock Freq. (MHz):   2100
  BDFID:                   50688
  Internal Node ID:        7
  Compute Unit:            304
  SIMDs per CU:            4
  Shader Engines:          32
  Shader Arrs. per Eng.:   1
  WatchPts on Addr. Ranges:4
  Coherent Host Access:    FALSE
  Features:                KERNEL_DISPATCH
  Fast F16 Operation:      TRUE
  Wavefront Size:          64(0x40)
  Workgroup Max Size:      1024(0x400)
  Workgroup Max Size per Dimension:
    x                        1024(0x400)
    y                        1024(0x400)
    z                        1024(0x400)
  Max Waves Per CU:        32(0x20)
  Max Work-item Per CU:    2048(0x800)
  Grid Max Size:           4294967295(0xffffffff)
  Grid Max Size per Dimension:
    x                        4294967295(0xffffffff)
```

```
    y                        4294967295(0xffffffff)
    z                        4294967295(0xffffffff)
  Max fbarriers/Workgrp:   32
  Packet Processor uCode:: 132
  SDMA engine uCode::      19
  IOMMU Support::          None
  Pool Info:
    Pool 1
      Segment:                 GLOBAL; FLAGS: COARSE GRAINED
      Size:                    201310208(0xbffc000) KB
      Allocatable:             TRUE
      Alloc Granule:           4KB
      Alloc Alignment:         4KB
      Accessible by all:       FALSE
    Pool 2
      Segment:                 GLOBAL; FLAGS: EXTENDED FINE GRAINED
      Size:                    201310208(0xbffc000) KB
      Allocatable:             TRUE
      Alloc Granule:           4KB
      Alloc Alignment:         4KB
      Accessible by all:       FALSE
    Pool 3
      Segment:                 GLOBAL; FLAGS: FINE GRAINED
      Size:                    201310208(0xbffc000) KB
      Allocatable:             TRUE
      Alloc Granule:           4KB
      Alloc Alignment:         4KB
      Accessible by all:       FALSE
    Pool 4
      Segment:                 GROUP
      Size:                    64(0x40) KB
      Allocatable:             FALSE
      Alloc Granule:           0KB
      Alloc Alignment:         0KB
      Accessible by all:       FALSE
  ISA Info:
    ISA 1
      Name:                    amdgcn-amd-amdhsa--gfx942:sramecc+:xnack-
      Machine Models:          HSA_MACHINE_MODEL_LARGE
      Profiles:                HSA_PROFILE_BASE
      Default Rounding Mode:   NEAR
      Default Rounding Mode:   NEAR
      Fast f16:                TRUE
      Workgroup Max Size:      1024(0x400)
      Workgroup Max Size per Dimension:
        x                        1024(0x400)
        y                        1024(0x400)
        z                        1024(0x400)
      Grid Max Size:           4294967295(0xffffffff)
      Grid Max Size per Dimension:
        x                        4294967295(0xffffffff)
```

```
      y                        4294967295(0xffffffff)
      z                        4294967295(0xffffffff)
    FBarrier Max Size:       32
```

Hangs or slow multi-GPU runs and IOMMU

If a multi-GPU ROCm run hangs or crawls, the host's IOMMU configuration is a common culprit. The correct setting is platform-specific - match your hardware instead of blindly disabling it:

- **Modern Instinct (MI300X, MI325X, MI350X, MI355X):** AMD's system-acceptance guides want the IOMMU **enabled** in BIOS and running in pass-through mode on the kernel command line - `iommu=pt` together with `amd_iommu=on` (or `intel_iommu=on` on Intel hosts). MI350X/MI355X (ROCm 7.0.1+) explicitly list `IOMMU enabled` + `iommu=pt`.
- **MI100 / MI200 (MI210/MI250/MI250X):** AMD's qualified BIOS tables typically set the NBIO **IOMMU to Disable**, with `iommu=pt` called out only for hosts with 256+ logical CPUs (so X2APIC can address every core).
- **Legacy last resort:** on older systems where neither pass-through nor the BIOS default helped, fully disabling it via `amd_iommu=off` (or the softer `iommu=soft`) was the historical workaround - see AMD's IOMMU advisory for multi-GPU environments and this issue report (that particular case was actually on NVIDIA A6000s). This is an invasive, system-wide boot change, and current AMD guidance prefers pass-through over a full disable.

These are boot-time kernel parameters set in `/etc/default/grub` (e.g. `GRUB_CMDLINE_LINUX="... iommu=pt"`; the grub config path varies by OS), followed by regenerating grub and rebooting - so they need admin access.

The per-GPU `IOMMU Support` line in the `rocminfo` output above reflects what the runtime actually sees.

AMD GPUs Performance

As I haven't had a chance to do any serious work with AMD GPUs, just sharing links for now.

- AMD Instinct MI300X system optimization
- AMD Instinct MI300X workload optimization

Profilers

omniperf - Advanced Profiling and Analytics for AMD Hardware - e.g. can plot a roofline performance of your AMD accelerator and many other things.

CPU

This chapter is short because GPUs dominate ML work - but the CPU's share has been growing, and unlike the GPU it's easy to under-provision without noticing.

Originally Machine Learning workloads didn't use much CPU other than for data processing (text/audio/video).

Around 2025 CPUs started to get more use for RAG workloads which need to perform database queries.

In 2026 the load on CPUs started increasing due to AI tool calling in RL workloads. These workloads may execute a variety of computer programs, which may perform validation of the generated data or code, compilation and execution of generated code, and many other things, and all of it typically running in a sandbox for security reasons. This need sometimes calls for dedicated CPU nodes to prevent stalling GPUs, since the CPU-cores co-located with the GPUs might prove insufficient.

How many CPU cores do you need for DataLoader work

In a typical training workload a DataLoader is what consumes most of the CPU cores.

Per 1 accelerator you need:

1. 1 CPU core per process that is tied to the accelerator
2. 1 CPU core for each `DataLoader` worker process - and typically you need 2-4 workers.

2 workers is usually plenty for LMs, especially if the data is already preprocessed.

If you need to do dynamic transforms, which is often the case with computer vision models or VLMs, you may need 3-4 and sometimes more workers.

The goal is to be able to pull from the `DataLoader` instantly, and not block the accelerator's compute, which means that you need to pre-process a bunch of samples for the next iteration, while the current iteration is running. In other words your next batch needs to take no longer than a single iteration accelerator compute of the batch of the same size.

Besides preprocessing if you're pulling dynamically from the cloud instead of local storage you also need to make sure that the data is pre-fetched fast enough to feed the workers that feed the accelerator furnace.

Multiply that by the number of accelerators, add a few cores for the Operation system (let's say 4).

If the node has 8 accelerators, and you have `num_workers`, then you need `8*(num_workers+1)+4`. If you're doing NLP, it'd be usually about 2 workers per accelerator, so `8*(2+1)+4` => 28 CPU cores. If you do CV training, and, say, you need 4 workers per accelerator, then it'd be `8(4+1)+4` => 44 CPU cores.

What happens if you have more very active processes than the total number of CPU cores? Some processes will get preempted (put in the queue for when CPU cores become available) and you absolutely want to avoid any context switching.

But modern cloud offerings typically have 50-100+ CPU-cores so usually there is no problem to have enough cores to go around.

See also Asynchronous DataLoader.

CPU needs for sandboxed code execution

To score what a model generated, RL tool-use workloads have to run it - a test suite, a compilation, a package install, a symbolic math check. That runs in a sandbox: an isolated environment, since generated code is untrusted by construction and one sample leaking state into the next corrupts the reward signal. Depending on how strong the isolation has to be,

this is a subprocess with resource limits, a container, a user-space kernel like gVisor, or a micro-VM like Firecracker.

None of that work touches the accelerator, and there is a lot of it - a batch of hundreds of trajectories each making dozens of tool calls means tens of thousands of short programs per training step.

Two things make it costlier than it looks:

1. Isolation taxes system calls, not arithmetic. gVisor adds no cost to CPU instructions, but routes every system call through its own user-space kernel, so in some situations file- and process-heavy work could slow by multiples - and a test run or a `pip install` is exactly that kind of work (gVisor performance guide). Micro-VMs avoid that tax but pay a per-sandbox boot instead, ~125ms for Firecracker.
2. It sits on the accelerators' critical path. A synchronous RL step can't proceed until the whole batch is done, so a CPU shortfall, a straggling sandbox, or a verifier hanging on a pathological output idles every GPU in the job.

Give the sandboxes their own cores, disjoint from the ones the training process and its `DataLoader` workers run on. A child process inherits its parent's allowed cores, so this has to be set on the sandbox itself, with `os.sched_setaffinity` or the container runtime's cpuset. The same section shows how the trainer side of that split is done. Cap the concurrency of the expensive phases like dependency installs and test runs, and put a timeout on everything - a generated infinite loop will otherwise hold a core until something kills it. At scale this is what pushes sandboxes onto dedicated CPU nodes.

footnote: A Nebius agentic-RL writeup (2026-09) measured this on a Terminal-Bench workload, GLM-5.2 on 32 B300s. With the number of in-flight trajectories held at 1,024, assigning the sandboxes deterministic non-overlapping CPU sets cut batch-collection time from 332.1s to 303.0s, with no change to the model or the GPU configuration.

CPU offload

Some frameworks, like DeepSpeed and FSDP can offload some compute work to CPUs without creating a bottleneck. In which case you'd want additional CPU-cores beyond what you already use.

NUMA affinity

See NUMA affinity.

Hyperthreads

Hyper-Threads double the CPU cores number, by virtualizing each physical core into 2 virtual ones, allowing 2 threads to use the same CPU core at the same time. Depending on the type of workload this feature may or may not increase the overall performance. Intel, the inventor of this technology, suggests a possible 30% performance increase in some situations.

See also To enable Hyper-Threads or not.

CPU Memory

This is a tiny chapter, since usually there are very few nuances one needs to know about CPU memory - which is a good thing!

Most of the ML workload compute happens on GPUs, but typically there should be at least as much CPU memory on each node as there is on the GPUs. So, for example, if you're on a H100 node with 8x 80GiB GPUs, you have 640GiB of GPU memory. Thus you want at least as much of CPU memory. Most recent high end cloud packages usually come with 1-2TiBs of CPU memory.

What CPU memory is needed for in ML workloads

- Loading the model weights, unless they are loaded directly onto the GPUs - this is usually a transitory memory usage that goes back to zero once the model has been moved to GPUs.
- Saving the model weights. In some situations each GPU writes its own checkpoint directly to the disk, in other cases the model is recomposed on the CPU before it's written to disk - this too is a transitory memory usage.
- Possible parameter and optimizer state offloading when using frameworks like DeepSpeed. In which case quite a lot of CPU memory might be needed.
- Activations calculated in the `forward` pass, and which need to be available for the `backward` path can also be offloaded to CPU, rather than discarded and then recomputed during the backward pass to save the unnecessary overhead
- `DataLoader` is usually one of the main users of CPU memory and at times it may consume very large amounts of memory. Typically there are at least 2x 8 DL workers running on each node, so you need enough memory to support at least 16 processes each holding some data. For example, in the case of streaming data from the cloud, if the data shards are large, these processes could easily eat up hundreds of GBs of CPU memory.
- The software itself and its dependent libraries uses a bit of CPU memory, but this amount is usually negligible.

Things to know

- If the `DataLoader` uses HF `datasets` in `mmap` mode the Resident memory usage may appear to be using a huge amount of CPU memory as it'll try to map out the whole datasets to the memory. Except this is misleading, since if the memory is needed elsewhere the OS will page out any unneeded mmap'ed pages back to the system. You can read more about it here. This awareness, of course, applies to any dataset using `mmap`, I was using HF `datasets` as an example since it's very widely used.

Storage: File Systems and IO

Machine Learning IO needs

For training workloads there are 3 distinct IO needs:

1. You need to be able to feed the DataLoader fast - (super fast read, don't care about fast write) - requires sustainable load for hours and days
2. You need to be able to write checkpoints fast - (super fast write, fastish read as you will be resuming a few times) - requires burst writing - you want super fast to not block the training for long (unless you use some sort of cpu offloading to quickly unblock the training)
3. You need to be able to load and maintain your codebase - (medium speed for both reading and writing) - this also needs to be shared since you want all nodes to see the same codebase - as it happens only during the start or resume it'll happen infrequently

As you can see these 3 have very different requirements both on speed and sustainable load, and thus ideally you'd have 3 different filesystems, each optimized for the required use case.

If you have infinite funds, of course, get a single super-fast read, super-fast write, that can do that for days non-stop. But for most of us, this is not possible so getting 2 or 3 different types of partitions where you end up paying much less is a wiser choice.

For inference, it's mostly about startup time to be able to start serving inference asap:

1. Reading the checkpoint fast
2. Loading the codebase fast

For batched inference, you'd have additionally:

1. Reading prompts
2. Writing generated outputs

If this is done asynchronously to generation then this IO overhead can be completely hidden, since typically generation will be much slower than loading the text it generates from and writing the generated text back to the disk.

If you do KV-cache offloading to disk, this would be another important IO use-case for inference. You will need to be able to seek fast to get to the desired KV-cache index and less crucial writing it out if the latter is done asynchronously.

Glossary

- IOPS: Input/Output Operations Per Second
- NSD: Network Shared Disk

Concepts

Here are a few key storage-related concepts that you likely need to be familiar with:

Queue Depth

Queue depth (or **IO depth**) is the number of IO requests that can be queued at one time on a storage device controller. If more IO requests than the controller can queue are being sent the OS will usually put those into its own queue.

On Linux the local block devices' queue depth is usually pre-configured by the kernel. For example, if you want to check the max queue depth set for `/dev/sda` you can `cat /sys/block/sda/queue/nr_requests`. To see the current queue depth

of a local device run `iostat -x` and watch for `aqu-sz` column. (`apt install sysstat` to get `iostat`.)

A storage device serves many requests at once - an NVMe drive reads from many flash chips simultaneously, a network file system (usually) from many storage servers. Think of those as the device's workers. While there are fewer requests in progress than workers, each request goes straight to an idle worker, so latency stays flat while throughput climbs with every request added. Issue them one at a time instead and the device sits idle after each completion, waiting for the application to come back with the next request.

Once there are more requests than workers, throughput is capped and the surplus gets queued, so latency grows with the depth.

Direct vs Buffered IO

Direct IO refers to IO that bypasses the operating system's caching buffers. This corresponds to `O_DIRECT` flag in `open(2)` system call.

The opposite is the **buffered** IO, which is usually the default way most applications do IO since caching typically makes things faster.

When we run an IO benchmark it's critical to turn the caching/buffering off, because otherwise the benchmark's results will most likely be invalid. You normally won't be reading or writing the same file hundreds of times in a row. Hence most likely you'd want to turn the direct mode on in the benchmark's flags if it provides such.

In certain situations opening files with `O_DIRECT` may actually help to overcome delays. For example, if the training program logs to a log file (especially on a slow shared file system), you might not be able to see the logs for many seconds if both the application and the file system buffering are in the way. Opening the log file with `O_DIRECT` by the writer can get the reader to see the logged lines much sooner.

And it's worth knowing what `O_DIRECT` doesn't promise before building a workflow on it:

- it isn't durability. It "makes an effort to transfer data synchronously, but does not give the guarantees of the `O_SYNC` flag that data and necessary metadata are transferred" - if you need the bytes to survive a crash you want `fsync/fdatasync`, or `O_SYNC` in addition to `O_DIRECT`.
- it may impose alignment restrictions on the length and the address of your buffer and on the file offset, varying by file system and kernel version. A misaligned IO may fail with `EINVAL` or silently fall back to buffered IO - which is a very quiet way for a benchmark to stop measuring what you think it measures. Since Linux 6.1 `statx(2)` with `STATX_DIOALIGN` will tell you the actual requirements.
- mixing `O_DIRECT` and normal buffered IO on the same file, especially on overlapping regions, is explicitly discouraged - and an `O_DIRECT` writer with a `tail -f` reader is precisely that pairing.

For the logging problem, though, try the application's own buffering first - it's usually the layer that's actually sitting on your lines, and it's much cheaper to change: `python -u` or `PYTHONUNBUFFERED=1` for Python, `flush=True` on the `print` call, `stdbuf -oL` for the C stdio programs in a pipeline, like `awk` or `cut`, which have no flushing flag of their own. `O_DIRECT` does nothing for this, because it bypasses the kernel's page cache and not your process' own buffer - a line still sitting in the application's buffer hasn't been written at all yet, whatever flags the file was opened with.

Here is a quick demonstration. `awk` is a plain C stdio program, which is exactly the kind of thing `stdbuf` can fix - the trailing `cat` is only there to make `awk`'s stdout a pipe rather than a terminal:

```
{ echo 1; sleep 3; echo 2; } | awk '{print}' | cat
```

Both lines appear together after 3 seconds. `awk` isn't refusing to print, it's printing into its own buffer - when stdout isn't a terminal, stdio switches from line buffering to a block buffer of several kilobytes and writes only when that buffer fills or the program exits. Add `stdbuf -oL` and `1` shows up immediately:

```
{ echo 1; sleep 3; echo 2; } | stdbuf -oL awk '{print}' | cat
```

Point `awk` at your terminal instead, by removing `| cat`, and stdio line-buffers by default, so both versions behave identically and there is nothing to fix - the buffering only bites once you pipe into another program or redirect into a log file, which is precisely when you can no longer watch it happening.

`stdbuf` presets the buffering mode that C stdio reads at startup, so it only reaches programs that leave that decision to stdio. `perl` and `python` both manage their own and will ignore it completely, as will Go binaries - for those the knob has to be inside the program: `$|=1` for perl, `-u` or `PYTHONUNBUFFERED=1` for python.

Synchronous vs asynchronous IO

In synchronous IO the client submits an IO request and wait for it to be finished before submitting the next IO request to the same target device.

In asynchronous IO the client may submit multiple IO requests one after another without waiting for any to finish first. This requires that the target device can queue up multiple IO requests.

Sequential vs Random access IO

Sequential access IO is when you read blocks of data one by one sequentially (think a movie). Here are some examples:

- reading or writing a model's checkpoint file all at once
- loading a python program
- installing a package

Random access IO is when you're accessing part of a file at random. Here are some examples:

- database querying
- reading samples from a pre-processed dataset in a random fashion
- moving around a file using `seek`

mmap vs sequential dataset reads

A dataset that looks like one large file may still end up doing random IO. Hugging Face `datasets` keeps its data in Apache Arrow files - a binary columnar format whose on-disk bytes are already laid out the way they will be used in memory, so a row can be read where it lies with no parsing step. That is what makes memory-mapping the default (`load_from_disk`, or `load_dataset` without `keep_in_memory`): the file is mapped into the address space and a row access becomes a page fault instead of a `read` call.

footnote: The resident memory this mapping shows in `top` is not a leak - the OS can reclaim unreferenced pages - see CPU memory.

If you count `read()` syscalls - with `strace -c`, or by grepping a trace - a mapped dataset looks almost idle. Each mapped run below issued 2 `read` calls for the whole 1GiB; the bytes still moved, but they show up as major page faults and in `/proc/self/io` `read_bytes` instead.

Measured on 2026-09 with `datasets` 4.0.0: 1GiB of text as 32768 rows of 32KiB in three Arrow shards, page cache dropped with `posix_fadvise(..., POSIX_FADV_DONTNEED)` before each run. The first row is the floor - the same three files read start to finish with ordinary `read()` calls of 1MiB each. The other two go through the mapped dataset: `for row in ds`, then `ds[i]` in shuffled order.

How the 1GiB was read	Local NVMe	Slowdown	Lustre	Slowdown
sequential `read()`, whole file	0.14s	0%	1.87s	0%

How the 1GiB was read	Local NVMe	Slowdown	Lustre	Slowdown
mmap, rows in order	0.53s	279%	3.11s	66%
mmap, rows shuffled	0.78s	457%	94.2s	4_937%

- The slowdown column is `t / t_sequential - 1` within each file system.
- `mmap-io-bench.py` created this table - edit the two paths at the top of the script before running. In this table the benchmarks filesystem was Lustre.

The three rows are the three ways a pipeline usually touches Arrow shards: read the file through once, walk a mapped dataset in order, or index rows in shuffled training order. Match your target job to a row - the bullets below say when the slowdown is acceptable and when you need a different read path or node-local shards instead of a network fs.

1. **Consuming a shard in order** - checksums, format conversion, one-pass tokenization. Read the file with ordinary `read`s of a MiB or so and do not map it. This is why it's not enough to keep datasets in large files. Additionally every reader has to consume them in large sequential chunks to get high performance.
2. **`for row in ds` over a mapped dataset** - on local NVMe the mapping costs 279% and still finishes a GiB in half a second, so leave it. On Lustre it costs 66% and every page comes over the network, so copy the shards to node-local disk first if this loop feeds training.
3. **Shuffled `ds[i]` over a mapped dataset** - ordinary shuffled training. On Lustre this took 94s per GiB, 50x the sequential read of the same bytes. Do not point it at a network file system. Stage the split on node-local NVMe and map it there, or pre-shuffle into epoch shards and read those in order, or stream and shuffle within a buffer. `keep_in_memory=True` removes the faults too, but trades them for RAM that every DataLoader worker pays again.

footnote: Reading the shards through sequentially before the loop - `cat shard > /dev/null`, or a `cp` - leaves them in the page cache, and the shuffled case then faults against RAM: on Lustre 98.24s from cold became a 1.86s pre-read plus a 1.03s loop. This holds only while the shards fit in the page cache, so it is a remedy for a split that fits in RAM rather than a replacement for staging on local NVMe. The mapping hints do not help and are not reachable through `datasets` in any case - `MAP_POPULATE` prefaults the whole mapping, `MADV_SEQUENTIAL` only widens readahead, and `pyarrow`'s `memory_map` passes neither.

Misreported file size

I have noticed some distributed file systems, like Lustre, may report incorrect file sizes if the files got offloaded and haven't been "rehydrated". I haven't seen this problem with Weka or GPFS. A proper distributed file system client should always report the real file size even if the contents of the file have been offloaded, and then automatically re-hydrate the file when it's being read.

If you're unlucky to deal with such a broken file system client, you can get a rough idea of the real file sizes using `du --apparent-size`, but beware that it may over-report the size if there is fragmentation, file sparsity and other reasons. `df` will still report incorrect file sizes, since it doesn't have a similar flag to `du`.

If you have to force re-hydration you can run something like:

```
find /mountpoint/ -type f -exec cat {} >/dev/null \;
```

and then both `du` and `df` will report correct file sizes, except the above command may take a really long time to run if you have hundreds GBs of data.

If at all possible, avoid using file systems which can't handle such a fundamental need as reporting correct file sizes, because when this occurs you may be unaware that your partition is close to being full. For example, it may report being

5% full when it's 95% full.

Which file system to choose

Distributed Parallel File Systems are the fastest solutions

Distributed parallel file systems dramatically improve performance where hundreds to thousands of clients can access the shared storage simultaneously. They also help a lot with reducing hotspots (where some data pockets are accessed much more often than others).

The 3 excellent performing parallel file systems that I had experience with are:

- GPFS (IBM), recently renamed to IBM Storage Scale, and before that it was called IBM Spectrum Scale.
- WekaIO
- Lustre FS (Open Source) (Wiki)

These solutions have been around for 2+ decades. They are POSIX-compliant. These are also not trivial to create - you have to setup a whole other cluster with multiple cpu-only VMs dedicated exclusively for those filesystems - only then you can mount those. As compared to weaker cloud-provided "built-in" solutions which take only a few screens of questions to answer in order to activate. And when creating the storage cluster there is a whole science to which VMs to choose for which functionality. For example, here is a Lustre guide on GCP.

case study: At JeanZay HPC (France) in 2021 we were saving 2.3TB checkpoint in parallel on 384 processes in 40 secs! This is insanely fast - and it was GPFS over NVME drives.

NASA's cluster has a long long list of gotchas around using Lustre.

Some very useful pros of GPFS:

- If you have a lot of small files, you can easily run out of inodes (`df -i` to check). GPFS 5.x never runs out of inodes, it dynamically creates more as needed
- GPFS doesn't have the issue Lustre has where you can run out of disk space at 80% if one of the sub-disks got full and wasn't re-balanced in time - you can reliably use all 100% of the allocated storage.
- GPFS doesn't use a central metadata server (or a cluster of those) which often becomes a bottleneck when dealing with small files. Just like data, metadata is handled by each node in the storage cluster.
- GPFS comes with a native NSD client which is superior to the generic NFS client, but either can be used with it.
- One can build a multi-tier system. So for example, Tier 1 is usually made from NVME drives and Tier 2 usually uses some cloud storage system. So when the Tier 1 capacity gets low, files that haven't been accessed in some time, get auto-moved to the cloud storage. So for example your Tier 1 could be 100TB, and Tier 2 could be 1PB. This approach saves a lot of money, since 1PB of cloud storage is significantly cheaper than 1PB of NVME drives.
- Data protection can use various RAID approaches. Typically striping is used to save costs.

Weka is quite similar to GPFS in features and performance. The main difference would be the licensing cost you can negotiate with either provider. A big part of your cost will be in the cost of the VMs required to run the system - e.g. if you have a lot of small files you'd want many VMs to quickly deal with meta-data.

Other parallel file systems I don't yet have direct experience with:

- BeeGFS
- DAOS (Distributed Asynchronous Object Storage) (Intel)
- NetApp
- VAST

Most clouds provide at least one implementation of these, but not all. If your cloud provider doesn't provide at least one of these and they don't have a fast enough alternative to meet your needs you should reconsider.

OK'ish solutions

There are many OK'ish solutions offered by various cloud providers. Benchmark those seriously before you commit to any. Those are usually quite decent for handling large files and not so much for small files.

case study: As of 2024-01 with GCP's Zonal FileStore over NFS solution `python -c "import torch"` takes 20 secs to execute, which is extremely slow! Once the files are cached it then takes ~2 secs. Installing a conda environment with a handful of prebuilt python packages can easily take 20-30 min! This solution we started with had been very painful and counter-productive to our work. This would impact anybody who has a lot of python packages and conda environments. But, of course, GCP provides much faster solutions as well.

Remote File System Clients

You will need to choose which client to use to connect the file system to your VM with.

The most common choice is: NFS - which has been around for 4 decades. It introduces an additional overhead and slows things down. So if there is a native client supported by your VM, you'd have an overall faster performance using it over NFS. For example, GPFS comes with an NSD client which is superior to NFS.

File Block size

If the file system you use uses a block size of 16MiB, but the average size of your files is 16KiB, you will be using 1,024 times more disk space than the actual use. For example, you will see 100GiB of disk space used when the actual disk space will be just 100MiB.

footnote: On Linux the native file systems typically use a block size of 4KiB.

So often you might have 2 very different needs and require 2 different partitions optimized for different needs.

1. thousands to millions of tiny files - 4-8KiB block size
2. few large files - 2-16MiB block size

case study: Python is so bad at having tens of thousand of tiny files that if you have many conda environments you are likely to run of inodes in some situations. At JeanZay HPC we had to ask for a special dedicated partition where we would install all conda environments because we kept running out of inodes on normal GPFS partitions. I think the problem is that those GPFS partitions were configured with 16MiB block sizes, so this was not a suitable partition for 4KiB-large files.

The good news is that modern solutions are starting to introduce a dynamic block size. For example, the most recent GPFS supports sub-blocks. So, for example, it's possible to configure GPFS with a block size of 2MiB, with a sub-block of 8KiB, and then the tiny files get packed together as sub-blocks, thus not wasting too much disk space.

Distributed storage servers proximity to clients

The cluster that uses a shared distributed storage should have the storage servers places close to the cluster that uses those servers. If the VMs running the storage servers are located many hops (switches) away, the IO latency can be high and the interactive use of the storage can be frustratingly slow. Think any interactions with metadata servers as an example, when you try to run `du` and other tools that access metadata of many files.

So if you have control ask the cloud provider to give you the cpu-only storage servers VMs allocated as close as possible to your accelerator VMs network-distance-wise.

Cloud shared storage solutions

Here are shared file system storage solutions made available by various cloud providers:

- GCP
- Azure
- AWS

Local storage beats cloud storage

While cloud storage is cheaper the whole idea of fetching and processing your training data stream dynamically at training time is very problematic with a huge number of potential issues around it.

Same goes for dynamic offloading of checkpoints to the cloud.

It's so much better to have enough disk space locally for data loading.

For checkpointing there should be enough local disk space for saving a checkpoint in a fast and reliable way and then having a crontab job or a slurm job to offload it to the cloud. Always keep the last few checkpoints locally for a quick resume, should your job crash, as it'd be very expensive to wait to fetch the checkpoint from the cloud for a resume.

case study: we didn't have a choice and had to use cloud storage for dataloading during IDEFICS-80B training as we had barely any local storage and since it was multimodal data it was many TBs of data. We spent many weeks trying to make this solution robust and it sucked at the end. The biggest issue was that it was very difficult at the time to keep track of RNG state for the DataSampler because the solution we used, well, didn't bother to take care of it. So a lot of data that took a lot of time to create was wasted (not used) and a lot of data was repeated, so we didn't have a single epoch of unique data.

In some situations people find good solutions for working with cloud-based datasets, I personally haven't had a smooth experience yet and that's why I advocate local storage. If you found a good streaming solution that can properly resume without losing data and repeating the same data, doesn't require huge local workers then it might work OK.

What makes that work is layout. Streaming pulls whole objects out of cloud object storage - an S3-style bucket, whose API hands back an entire object rather than letting you read a piece of one - so it pays off only when the dataset is already large shards consumed front to back. That is the same shape mmap vs sequential dataset reads found fastest on a shared file system, so a dataset in that shape needs no second copy there - the bucket can feed training directly.

That whole-object interface doubles as a guardrail, since it makes seeking inside an object or memory-mapping one impossible. Mounting the bucket so it appears as an ordinary directory removes the guardrail: the random-access and memory-mapping patterns that section clocked at 50x the sequential read on a network fs (Lustre in that example) start working again, against a backend that was never designed for them.

Beware that you're often being sold only 80% of the storage you pay for

There is a subtle problem with distributed shared storage used on compute nodes. Since most physical disks used to build the large file systems are only 0.3-2TB large, any of these physical disks can get full before the combined storage gets full. And thus they require constant rebalancing so that there will be no situation where one disk is 99% full and others are only 50% full. Since rebalancing is a costly operation, like most programming languages' garbage collection, it happens infrequently. And so if you run `df` and it reports 90% full, it's very likely that any of the programs can fail at any given time.

From talking to IO engineers, the accepted reality (that for some reason is not being communicated to customers) is that only about 80% of distributed large storage is reliable.

Which means that if you want to have 100TB of reliable cloud storage you actually need to buy 125TB of storage, since 80% of that will be 100TB. So you need to plan to pay 25% more than what you provisioned for your actual needs. I'm not sure why the customer should pay for the technology deficiency but that's how it is.

For example, GCP states that only 89% can be used reliably, albeit more than once the storage failed already at 83% for me there. Kudos to Google to even disclosing this as a known issue, albeit not at the point of where a person buys the storage. As in - we recommend you buy 12% more storage than you actually plan to use, since we can only reliably deliver 89% of it.

I also talked to Sycomp engineers who provide managed IBM Storage Scale (GPFS) solutions, and according to them GPFS doesn't have this issue and the whole 100% can be reliably used.

Also on some setups if you do backups via the cloud provider API (not directly on the filesystem), they might end up using the same partition, and, of course, consume the disk space, but when you run `df` it will not show the real disk usage - it may show usage not including the backups.

Whatever storage solution you pick, ask the provider how much of the storage can be reliably used, so that there will be no surprises later.

Beware that on some cloud providers backups use the same partition they backup

This makes no sense to me but with some providers when you make a back up of a partition using their tools, the back up will use space on that same partition. And on some of those providers you won't even know this happened until you run out of disk space when you really used 30% of the partition you allocated. On those providers running `df` is pointless because it'll tell you the free disk space, but it won't include any back ups in it. So you have no idea what's going on.

If you start making a backup and suddenly everything fails because all processes fail to write but `df` reports 30% usage, you will now know why this happened. Snapshots too use the same partition.

So say you paid for a 100TB partition and you used up 95TB and now you want to back it up - well, you can't - where would it put 95TB of data if it has 5TB of data left even if it compresses it.

As I discover specific solution that have this unintuitive behavior I will add pointers to how you can see the actual disk usage:

- GCP FileStore (but it doesn't work for Basic Tier)

Don't forget the checksums

When you sync data to and from the cloud make sure to research whether the tool you use checks the checksums, otherwise you may end up with corrupt during transmission data. Some tools do it automatically, others you have to enable this feature (since it usually comes at additional compute cost and transmission slowdown). Better slow, but safe.

These are typically MD5 and SHA256 checksums. Usually MD5 is sufficient if your environment is safe, but if you want the additional security do SHA256 checksums.

Benchmarks

Time is money both in terms of a developer's time and model's training time, so it's crucial that storage IO isn't a bottleneck in your human and compute workflows.

In the following sections we will discuss various approaches to figuring out whether the proposed storage solution satisfies your work needs.

Metrics

The three main storage IO metrics one typically cares for are:

1. Throughput or Bandwidth (bytes per second - can be MBps, GBps, etc.)
2. IOPS (Input/output operations per second that a system can perform)
3. Latency (msecs or usecs)

- *IOPS* measures how many input and/or output operations a given storage device or a cluster can perform per second. Typically read and write IOPS won't be the same. And for many systems it'll also depend on

whether the operation is sequential or random. So a storage system will have 4 different IOPS rates:

1. IOPS of random reads
2. IOPS of random writes
3. IOPS of sequential reads
4. IOPS of sequential writes

- *Throughput* refers to how much data can be processed per second.

IOPS vs. Throughput

- when you deal with small files high IOPS is important.
- when you deal with large files high throughput is important.

IOPS correlates to Throughput via block size: `Throughput = IOPS * block_size`

Thus given a fixed IOPS - the larger the block size that the system can read or write the bigger the throughput will be.

And since there are 4 IOPS categories, correspondingly there are 4 throughput values to match.

Latency: is the delay between the moment the instruction to transfer data is issued and when the response to that instruction arrives.

Typically the more distance (switches, relays, actual distance) the packet has to travel the bigger the latency will be.

So if you have a local NVME drive your read or write latency will be much shorter as compared to reading or writing to a storage device that is located on another continent.

fio

fio - Flexible I/O tester is a commonly used IO benchmarking tool, which is relatively easy to operate. It has many options which allow you to emulate pretty much any type of a load and it provides a very detailed performance report.

First install `fio` with `apt install fio` or however your package manager does it.

Here is an example of a read benchmark:

```
base_path=/path/to/partition/
fio --ioengine=libaio --filesize=16k --ramp_time=2s --time_based --runtime=3m --numjobs=16 \
--direct=1 --verify=0 --randrepeat=0 --group_reporting --unlink=1 --directory=$base_path  \
--name=read-test --blocksize=4k --iodepth=64 --readwrite=read
```

Here 16 concurrent read threads will run for 3 minutes. The benchmark uses a block size of 4KiB (typical for most OSes) with the file size of 16KiB (a common size of most Python files) in a sequential reading style using non-buffered IO. So this particular set of flags will create a good proxy benchmark for how fast you can import Python modules on 16 concurrent processes.

case study: on one NFS setup we had `python -c "import torch"` taking 20 seconds the first time it was run, which is about 5-10x slower than the same test on a normal NVME drive. Granted once the files were cached the loading was much faster but it made for a very painful development process since everything was slow.

good read: Fio Output Explained - it's an oldie but is still a goodie - if you have a more up-to-date write up please send me a link or a PR.

Important: if you don't use the `--unlink=1` flag make sure to delete `fio`'s work files between different benchmarks - not doing so can lead to seriously wrong reports as `fio` will reuse files it prepared for a different benchmark which must not be re-used if the benchmark parameters have changed. Apparently this reuse is an `fio` feature, but to me it's a bug since I didn't know this nuance and got a whole lot of invalid reports because of it and it took awhile to realize they were wrong.

Going back to the benchmark - the parameters will need to change to fit the type of the IO operation you care to be fast - is it doing a lot of pip installs or writing a checkpoint on 512 processes, or doing a random read from a parquet file - each benchmark will have to be adapted to measure the right thing.

At the beginning I was manually fishing out the bits I was after, so I automated it resulting in fio-scan benchmark that will run a pair of read/write benchmarks on 16KiB, 1MiB and 1GiB file sizes each using a fixed 4KiB block size (6 benchmarks in total). It uses a helper fio-json-extract.py to parse the log files and pull out the average latency, bandwidth and iops and report them in a nicely formatted markdown table.

Here is how to run it:

```
git clone https://github.com/stas00/ml-engineering/
cd ml-engineering
cd storage

path_to_test=/path/to/partition/to/test
./fio-scan $path_to_test
```

Adapt `path_to_test` to point to the partition path you want to benchmark.

note: `fio-scan` calls `python ./fio-json-extract.py`, so if it fails check what `python` resolves to - on many systems only `python3` exists. Edit `fio-scan` to point at the right interpreter.

Here is an example of this IO scan on my Samsung SSD 980 PRO 2TB NVME drive, run on 2023-12-20 against a `/mnt/nvme0` partition:

- filesize=16k read

lat msec	bw MiBps	IOPS	jobs
4.0	1006.3	257614	16

- filesize=16k write

lat msec	bw MiBps	IOPS	jobs
3.2	1239.1	317200	16

- filesize=1m read

lat msec	bw MiBps	IOPS	jobs
1.7	2400.1	614419	16

- filesize=1m write

lat msec	bw MiBps	IOPS	jobs
2.1	1940.5	496765	16

- filesize=1g read

lat msec	bw MiBps	IOPS	jobs
1.4	2762.0	707062	16

- filesize=1g write

lat msec	bw MiBps	IOPS	jobs
2.1	1943.9	497638	16

As you can see this is a fast PCIe 4.0 NVMe drive if you want to use it as a baseline against, say, a network shared file system.

Usability perception IO benchmarks

Besides properly designed performance benchmarks which give you some numbers that you may or may not be able to appreciate there is a perception benchmark, and that is how does a certain functionality or a service feel. For example, when going to a website, does it feel like it's taking too long to load a webpage? or when going to a video service, does it take too long for the video to start playing and does it stop every few seconds to buffer the stream?

So with file system the questions are very simple - does it feel that it takes too long to install or launch a program? Since a lot of us live in the Python world, python is known to have thousands of tiny files which are usually installed into a virtual environment, with conda being the choice of many as of 2026-08.

In one of the environments we have noticed that our developers' productivity was really bad on a shared filesystem because it was taking up to 30min to install a conda environment with various packages needed for using a certain ML-training framework, and we also noticed that `python -c "import torch"` could take more than 20 seconds. This is about 5-10x slower than a fast local NVME-based filesystem would deliver. Obviously, this is bad. So I devised a perception test using `time` to measure the common activities. That way we could quickly tell if the proposed shared file system solution that we contemplated to switch to were significantly better. We didn't want a solution that was 2x faster, we wanted a solution that was 10x better, because having an expensive developer wait for proverbial paint to dry is not a good thing for a business.

So here is the poor man's benchmark that we used, so this is just an example. Surely if you think about the workflow of your developers you would quickly identify where things are slow and devise yours best fitting your needs.

note: To have a baseline to compare to do these timing tests on a recently manufactured local NVME. This way you know what the ceiling is, but beware that many shared file systems won't be able to match it.

Step 1. Install conda onto the shared file system you want to test if it's not there already.

```
export target_partition_path=/mnt/weka  # edit me!!!
mkdir -p $target_partition_path/miniconda3
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh -O
$target_partition_path/miniconda3/miniconda.sh
bash $target_partition_path/miniconda3/miniconda.sh -b -u -p $target_partition_path/miniconda3
rm -rf $target_partition_path/miniconda3/miniconda.sh
$target_partition_path/miniconda3/bin/conda init bash
bash
```

notes:

- adapt `target_partition_path` and the miniconda download link if you aren't on the x86 platform.
- at the end we launch a new `bash` shell for conda setup to take an effect, you might need to tweak things further if you're not a `bash` user - I trust you will know what to do.

Step 2. Measure conda install time (write test)

Time the creation of a new conda environment:

```
time conda create -y -n install-test python=3.12
```

```
real    0m22.790s
user    0m12.911s
sys     0m4.941s
```

Time the installation of some heavy pip packages:

```
conda deactivate
conda activate install-test
time pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu130
```

```
real    0m50.596s
user    0m28.060s
sys     0m4.398s
```

Please note that this test is somewhat skewed since it also includes the packages download in it and depending on your incoming network speed it could be super fast or super slow and could impact the outcome. But once the downloaded packages are cached, in the case of conda they are also untarred, so if you try to install the packages the 2nd time the benchmark will no longer be fair as on a slow shared file system the untarring could be very slow and we want to catch that.

I don't worry about it because usually when the file system is very slow usually you can tell it's very slow even if the downloads are slow, you just watch the progress and you can just tell.

If you do want to make this benchmark precise, you probably could keep the pre-downloaded conda packages and just deleting their untar'ed dirs:

```
find $target_partition_path/miniconda3/pkgs -mindepth 1 -type d -exec rm -rf {} +
```

in the case of `pip` it doesn't untar anything, but just caches the wheels it downloaded, so the `time pip install` benchmark can definitely be more precise if you run it the 2nd time (the first time it's downloaded, cached and installed, the second time it's installed from cache. So you could do:

```
conda create -y -n install-test python=3.12
conda activate install-test
pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu130
conda create -y -n install-test2 python=3.12
```

```
conda activate install-test2
time pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu130
```

As you can see here we time only the 2nd time we install the pip packages.

If you want a write test with no download mixed into it, copy a package you already have onto the file system under test and time that. Here is the `torch` package - 1.67GiB spread over 13019 files - copied onto a local NVMe drive and onto a shared Lustre mount:

```
$ export SITE_PACKAGES=$(python -c 'import site; print(site.getsitepackages()[0])')

$ time cp -a $SITE_PACKAGES/torch /tmp/torch-copy

real    0m2.026s
user    0m0.073s
sys     0m1.133s

$ time cp -a $SITE_PACKAGES/torch $target_partition_path/torch-copy

real    2m35.880s
user    0m0.266s
sys     0m9.262s
```

That is 77x slower, and a repeat run a few minutes earlier gave 2m30.0s, so it isn't a fluke. `user` and `sys` barely move between the two, so nearly all of that time is spent waiting on the file system - most of those 13019 files are small, and each one is a create that the metadata server has to answer.

Step 3. Measure loading time after flushing the memory and file system caches (read test)

```
sudo sync
echo 3 | sudo tee /proc/sys/vm/drop_caches
time python -c "import torch"
```

As you can see before we do the measurement we have to tell the OS to flush its memory and file system caches.

If you don't have `sudo` access you can skip the command involving `sudo`, also sometimes the system is setup to work w/o `sudo`. If you can't run the syncing and flushing of the file system caches you will just get incorrect results as the benchmark will be measuring the time to load already cached file system objects. To overcome this either ask your sysadmin to do it for you or simply come back in the morning while hopefully your file system caches other things and evicts the python packages, and then repeat the python one liner then with the hope those files are no longer in the cache.

Here is how to see the caching effect:

```
$ sudo sync
$ echo 3 | sudo tee /proc/sys/vm/drop_caches
$ time python -c "import torch"

real    0m2.438s
user    0m8.115s
```

```
sys     0m0.434s

$ time python -c "import torch"

real    0m1.333s
user    0m8.144s
sys     0m0.214s

$ sudo sync
$ echo 3 | sudo tee /proc/sys/vm/drop_caches
$ time python -c "import torch"

real    0m2.357s
user    0m8.177s
sys     0m0.368s
```

You can see that the first time it wasn't cached and took longer, then when I ran it the second time it was faster because everything was cached. And then I told the system to flush memory and file system caches and you can see it was slow again.

It helps to know what those seconds are made of. Two different kinds of IO go on during an import: a handful of large sequential reads for the shared objects in `torch/lib`, and a couple of thousand tiny operations - one open per module, plus the lookups that fail while Python probes candidate paths for each one. `strace` counts the second kind:

```
$ strace -cf -e trace=openat python -c "import torch"
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
100.00    0.005894           2      2339       922 openat
------ ----------- ----------- --------- --------- ----------------
100.00    0.005894           2      2339       922 total
```

Those 2339 opens - 922 of which failed, since Python probes several candidate paths per module - are a property of the package and the version installed. What each one takes is a property of the file system, and the gap is not subtle. Opening and closing a few thousand small files, on the local NVMe of the same node and on its shared Lustre mount:

open of one small file	local NVMe	Lustre
succeeds	11us	2.3ms
fails with `ENOENT`	2.4us	1.1ms

Apply those to the mix an `import torch` actually performs - 1417 opens that succeed and 922 that fail - and the opens alone go from about 0.02s on the local drive to about 4.3s on Lustre. A shared file system with per-open latency in that range is how a 1.3s import turns into the 20s reported in the case studies above. Repeating the Lustre measurement with everything already read changes nothing, either: what is being paid for is a metadata round trip, and the page cache has nothing to offer it.

The opens are an IOPS question and the bulk reads of the shared objects in `torch/lib` a throughput one, and a file system can be good at one and poor at the other.

I think it might be a good idea to do the memory and file system caching in the write tests again, since even there caching will make the benchmark appear faster than what it would be like in the real world where a new package is installed for the first time.

There is one more thing that even a cache-flushed read test doesn't capture. The first import after an installation compiles each `.py` file it touches into bytecode and writes the `.pyc` into `__pycache__`; every import after that reads the `.pyc` and skips the compile. Flushing the caches doesn't undo that, because the `.pyc` files are still on the disk - so a true first run after an install is slower than any of the numbers above, and a single throwaway `python -c "import torch"` is all it takes to leave that compiled state behind.

Another time I noticed that `git status` was taking multiple seconds. I use bash-git-prompt and it runs `git status` before every return of the prompt when inside a git repo clone, and it was becoming super sluggish and difficult to work. So I benchmarked `git status`:

```
git clone https://github.com/pytorch/pytorch
cd pytorch
time git status
```

and it was taking 3.7 secs on this slow file system and needed to be fixed (it was taking 0.02 secs on a local SSD). The good thing this actual perception benchmark was easy to pass to a sysadmin and them reproducing the problem instantly and then working on fixing it, while re-using this benchmark as a reference.

Yet, another time I noticed, `pytest` was taking forever to start, so I measured its collection and it indeed was very slow:

```
time pytest --disable-warnings --collect-only -q
```

So now you have a plethora of examples to choose from and I trust you will find your own use cases which are easy to reliably reproduce and use as a reference point for what feels good and what doesn't and which need to be fixed.

other tools

For a more rigorous, reproducible benchmark of a shared/parallel file system than the ad-hoc timings, two tools are worth knowing:

- IOR - the standard HPC parallel-IO benchmark. It launches many processes via MPI that read/write concurrently, so it measures the *aggregate* sequential and random bandwidth a distributed file system can sustain under realistic parallel load, across multiple backends (POSIX, MPI-IO, HDF5, ...). The same repo also ships `mdtest` (merged into `ior` in 2017), which instead benchmarks *metadata* performance - how many file/directory create/stat/remove operations per second the file system can handle. Metadata is frequently the real bottleneck for workloads with lots of small files, such as many small checkpoints or dataset shards.
- DLIO - a Deep-Learning I/O benchmark that emulates the data-loading pattern of ML training without needing any accelerator: it replaces the forward/backward compute with a `sleep` of equivalent duration and drives real data loaders (PyTorch `DataLoader`, `tf.data`) over synthetic datasets described via a YAML workload config. A run has three phases - generate synthetic data, run the benchmark, then post-process into a report. This makes it the closest proxy to "how will my storage behave during actual training", as opposed to the generic bandwidth/IOPS numbers IOR reports.

Published benchmarks

Here are some published IO benchmarks:

- MLPerf via MLCommons publishes various hardware benchmarks that measure training, inference, storage

and other tasks' performance. The storage results are the relevant ones here - that page always shows the current round, which advances a few times a year. Though I find the results are very difficult to make sense of - too many columns and no control whatsoever by the user, and each test uses different parameters - so how do you compare things.

Why pay for more storage when you can easily clean it up instead

Talking to a few storage providers I understood that many companies don't bother cleaning up and just keep on buying more and more storage. If you're not that company and want to keep things tidy in the following sections I will share how to easily prune various caches that many of us in the Python/PyTorch ecosphere use (and a lot of those will apply to other ecospheres).

HuggingFace Hub caches

The very popular HuggingFace Hub makes it super easy to download models and datasets and cache them locally. What you might not be aware of is that whenever a new revision of the model or a dataset is released, the old revisions remain on your disk - so over time you are likely to have a lot of dead weight.

The cached files are usually found at `~/.cache/huggingface`, but the least error-prone way to place them elsewhere is to set the `HF_HOME` environment variable before importing Hugging Face libraries:

```
export HF_HOME=/some/path/huggingface
mkdir -p "$HF_HOME"
```

The deprecated `HUGGINGFACE_HUB_CACHE` and library-specific variables such as `TRANSFORMERS_CACHE` may still be encountered in older setups.

The other solution that requires no environment variables, is to symlink your cache to another partition. You could do it for all of your caches:

```
mkdir -p /some/path
mv -nT ~/.cache /some/path/.cache
ln -s /some/path/.cache ~/.cache
```

Or move only the default Hugging Face cache and leave a symlink in its place. This script stops instead of merging with a pre-existing target or replacing an existing symlink:

```
#!/usr/bin/env bash
set -euo pipefail

source_dir="$HOME/.cache/huggingface"
target_dir="/some/path/huggingface"

mkdir -p "$(dirname "$source_dir")" "$(dirname "$target_dir")"

if [ -e "$target_dir" ] || [ -L "$target_dir" ]; then
    printf 'Target already exists: %s\n' "$target_dir" >&2
    exit 1
fi
```

```
if [ -L "$source_dir" ]; then
    printf 'Source is already a symlink: %s\n' "$source_dir" >&2
    exit 1
elif [ -e "$source_dir" ]; then
    mv "$source_dir" "$target_dir"
else
    mkdir -p "$target_dir"
fi

ln -s "$target_dir" "$source_dir"
```

The absent-source branch creates an empty target before linking it. If the target already contains a cache that should be reused, inspect it first and create the symlink manually rather than asking the script to merge directories.

Now that you know where the caches are, you could, of course, nuke the whole cache every so often, but if these are huge models and datasets, and especially if there was some preprocessing done for the latter - you really won't want to repeat those time consuming tasks again and again. So I will teach you how to use special tools provided by HuggingFace to do the cleanup.

The way revisions work on the HF hub is by pointing `main` to the latest revision of the files while keeping the old revisions around should anyone want to use the older revision for some reason. Chance are very high you always want the latest revision, and so here is how to delete all old revisions and only keeping `main`:

```
pip install -U "huggingface_hub"
hf cache prune
```

`hf cache prune` deletes every revision that no longer has a branch, tag, or pull-request ref pointing at it - which is precisely the old detached revisions you wanted gone - along with any `.incomplete` blobs left behind by interrupted downloads. Start with `--dry-run` to see the damage without deleting anything:

```
$ hf cache prune --dry-run
About to delete 30 unreferenced revision(s) and 9 incomplete download(s) (180.9G total).
  - dataset/fan-shu/instruct2thinking:
      4c8e57ed5f06425ce0d743cf168b906646f0a338 [(detached)] 609.7M
      ce82e962df666fc85657a3a871443009ec825b5f [(detached)] 259.7M
      d5a3ba55f8ac010ee0f1ac86314b49475a1e300a [(detached)] 226.2M
  - dataset/fan-shu/swe-instruct-trajectories-empty-think-inserted:
      84d55e1f36605fe990feb61a1a705032f0620015 [(detached)] 12.1G
  - model/Qwen/Qwen3.5-27B:
      b7ca741b86de18df552fd2cc952861e04621a4bd [(detached)] 55.6G
  - model/zai-org/GLM-5.2:
      f2263102df303b2faa54a6861a29d1770ce846c0 [(detached)] 1.5T
✓ Dry run: no files were deleted.
```

The total is reclaimable space after shared blobs are counted once; the per-revision sizes above can sum to much more. Without `--dry-run` it asks for confirmation before deleting; add `--yes` to skip the prompt when you run this from a cron job.

To find out where the space went before deleting anything, `hf cache ls` gives you per-repo totals and `hf cache ls --revisions` breaks it down per revision. It takes filters, which understand human sizes and durations, so you can go hunting for the big and the forgotten:

```
hf cache ls --revisions --filter "size>1GB" --filter "accessed>30d"
```

And to delete entire models or datasets rather than just their stale revisions, `hf cache rm` takes repo ids and revision hashes:

```
hf cache rm model/bert-base-cased dataset/glue
```

which pairs with `hf cache ls -q` - identifiers only, one per line - to delete by filter:

```
hf cache rm $(hf cache ls --filter "accessed>1y" -q) -y
```

If your cache isn't in the default location, every one of these takes `--cache-dir PATH`.

Additionally you will find that HF `datasets` have a `~/.cache/huggingface/datasets/downloads` dir which often will contain a ton of leftovers from datasets downloads and their preprocessing, including various lock files. On one setup I found literally a few millions of files there. So here is how I clean those up:

```
sudo find ~/.cache/huggingface/datasets/downloads -type f -mtime +3 -exec rm {} \+
sudo find ~/.cache/huggingface/datasets/downloads -type d -empty -delete
```

The first command leaves files that are younger than 3 days in place, in case someone is in the process of download/processing things and we don't want to swipe the carpet from under their feet.

As usual you may need to adjust the paths if you placed your caches elsewhere.

note: if your team uses `HF_HOME` to share the HF hub models/datasets/etc - the `$HF_HOME/token` will get shared as well, which works fine as long as ungated models are used. But if you want to access gated models you might run into problems there. Therefore you most likely want to not share the access token. You can fix that by adding something like:

```
export HF_TOKEN_PATH=~/.cache/hf_hub_token
```

(then put it into `~/.bashrc` to always work)

Now have each user run once:

```
hf auth login
```

which will ask them to add their access token from https://huggingface.co/settings/tokens - it'll save it under `~/.cache/hf_hub_token`.

Now each member of your team will have their unique token and the gated models approved for their HF hub user will

now be accessible by them.

Python package manager cleanups

conda and pip will pile up more and more files on your system over time. conda is the worst because it keeps the untarred files which consume an insane amount of inodes and make backups and scans slow. pip at least caches just the wheels (tarred files).

So you can safely nuke these dirs:

```
rm -rf ~/.cache/pip
rm -rf ~/anaconda3/pkgs/
```

Make sure to edit the last command if your conda is installed elsewhere.

Share caches in group environments

If you have more than 2 people working on the same system, you really want to avoid each person having their own cache of `pip`, `conda`, HF models, datasets and possibly other things. It is very easy to get each user's setup to point to a shared cache.

For example, let's say you make `pip` and `conda` caches under `/data/cache` like so:

```
mkdir /data/cache/conda
mkdir /data/cache/pip
chmod a+rwx /data/cache/conda
chmod a+rwx /data/cache/pip
```

now you just need to symlink from each user's local cache to this shared cache:

```
mkdir -p ~/.cache

rm -rf ~/.cache/pip
ln -s /data/cache/pip ~/.cache/pip

rm -rf ~/.conda/pkgs
ln -s /data/cache/conda/pkgs ~/.conda/pkgs
```

note that we wiped out the existing caches, but you could also move them to the shared cache instead - whatever works, you will want to periodically nuke those anyway.

So now when `pip` or `conda` will try to reach the user caches they will get redirected to the shared cache. If you have 20 people in the group that's 20x less files - and this is very important because conda pkg files are untarred and take up a huge amount of inodes on the disk.

So the only issue with this approach is file permissions. If user A installs some packages, user B might not be able to read or write them.

If this is an isolated cluster where there are no malicious users you can simply ask everybody to use `umask 000` in their `~/.bashrc` or even configuring this setting system-wide via `/etc/profile` or `/etc/bash.bashrc` and different other shell

config files if `bash` isn't your shell of choice.

Once `umask 000` is run, most files will be created with read/write perms so that all users can read/write each others files.

Of course, if you are using a sort of HPC, where many unrelated groups use the same cluster this won't work and then you would either use groups instead of making files read/write by all, with possibly `setgid` bit preset or using ACL . In any such environments there are always sysadmins so you can ask them how to setup a shared cache for your team and they will know what to do.

Additionally, recently some of these applications added tools to do the cleanup, e.g. for `conda` and `pip`:

```
conda clean --all -f -y
pip cache purge
```

General disk usage

Of course, sooner or later, your partition will get bigger and bigger, and you will probably want to understand where data is leaking. Typically you will need to find the users who contribute to the most of data consumption and ask them to do some cleanups.

So for example to find which users consume the most disk run:

```
sudo du -ahd1 /home/* | sort -rh
```

it will sort the data by the worst offenders. If you want to help them out you could go into their dirs and analyze the data a level deeper:

```
sudo du -ahd1 /home/*/* | sort -rh
```

or for a specific user `foo`:

```
sudo du -ahd1 /home/foo/* | sort -rh
```

You could also set disk usage quotas but usually this doesn't work too well, because depending on the workflows of your company some users need to generate a lot more data then others, so they shouldn't be punished for that with inability to do their work and have their job crash - which could have been run for many hours and all that work will be lost - so at the end of the day the company will be paying for the lost time.

Getting users to be aware of them using too much disk space can be a very difficult task.

Partition inodes limit

Also beware of inode usage, on some shared partitions on HPCs I have seen more than once cases where a job crashed not because there was no disk space left, but because the job used up the last inodes and the whole thing crashed.

To see inode usage, use `df -i`:

```
$ /bin/df -hi
```

```
Filesystem     Inodes IUsed IFree IUse% Mounted on
tmpfs             16M  1.9K   16M    1% /run
/dev/sda1         59M  4.1M   55M    7% /
```

`-h` formats huge numbers into human-readable strings.

So here you can see the `/` partition is using 7% of the total possible inodes.

Depending on the type of filesystem in some cases it's possible to add more inodes whereas in other cases it's not possible.

So as part of your monitoring of disk space you also need to monitor inode usage as a critical resource.

`/tmp` on compute nodes

Normally compute nodes will use `/tmp/` for temp files. The problem is on most set ups `/tmp` resides on the tiny `/` filesystem of each node (often <100GB) and since `/tmp/` only gets reset on reboot, this doesn't get cleaned up between SLURM jobs and this leads to `/tmp` running out of space and so when you try to run something that let's say untars a file you're likely to run into:

```
OSError: [Errno 28] No space left on device
```

The solution is to set in your SLURM launcher script.

```
export TMPDIR=/scratch
```

Now, the slurm job will use a much larger `/scratch` instead of `/tmp`, so plenty of temp space to write too.

footnote: while `/scratch` is quite common - the mounted local SSD disk mount point could be named anything, e.g. `/localssd` - it should be easy to see the right path by running `df` on one of the compute nodes.

You can also arrange for the SLURM setup to automatically clean up such folders on job's termination.

How to find users whose checkpoints consume a lot of disk space

Do you have a problem when your team trains models and you constantly have to buy more storage because huge model checkpoints aren't being offloaded to bucket storage fast enough?

Here is a one-liner that will recursively analyze a path of your choice, find all the checkpoints, sum up their sizes and print the totals sorted by the biggest user, so that you could tell them to clean up their act :) Just edit `/mypath` to the actual path:

```
find /mypath/ -type f -regextype posix-egrep -regex ".*\.(pt|pth|ckpt|safetensors)$" | \
perl -nle 'chomp; ($uid,$size)=(stat($_))[4,7]; $x{$uid}+=$size;
END { map { printf qq[%-10s: %7.1fTiB\n], (getpwuid($_))[0], $x{$_}/2**40 }
sort { $x{$b} <=> $x{$a} } keys %x }'
```

This produces an output like:

```
user_a   :     2.5TiB
user_c   :     1.6TiB
```

```
user_b   :     1.2TiB
```

Of course, you can change the regex to match other patterns or you can remove it altogether to measure all files:

```
find /mypath/ -type f | \
perl -nle 'chomp; ($uid,$size)=(stat($_))[4,7]; $x{$uid}+=$size;
END { map { printf qq[%-10s: %7.1fTiB\n], (getpwuid($_))[0], $x{$_}/2**40 }
sort { $x{$b} <=> $x{$a} } keys %x }'
```

If you want to exclude some sub-dirs efficiently:

```
find /mypath/ -regextype posix-egrep \
-type d -regex "/mypath/(exclude_a|exclude_b|exclude_c)" -prune -o \
-type f -regex ".*\.(pt|pth|ckpt|safetensors)$" -print | \
perl -nle 'chomp; ($uid,$size)=(stat($_))[4,7]; $x{$uid}+=$size;
END { map { printf qq[%-10s: %7.1fTiB\n], (getpwuid($_))[0], $x{$_}/2**40 }
sort { $x{$b} <=> $x{$a} } keys %x }'
```

hint: the second line tells `find` to skip folders matching the `/mypath/(exclude_a|exclude_b|exclude_c)` regex. Two subtleties: the regex has to match the folder's path exactly, because if you append a trailing `/.*` then `find` descends into the folder itself and prunes only its sub-folders - so any files sitting directly in it still get counted. And the `-print` is required, because `-prune` is an action and without a second one `find` prints the pruned folders too, which then get `stat`ed and added to somebody's total. Adapt to your use case as needed.

How to automatically delete old checkpoints

Continuing from finding the users whose checkpoints consume the most disk space, here is how to automatically delete old checkpoints (e.g. those older than 30 days).

First try to ensure the candidates are indeed good to delete:

```
find /mypath/ -regextype posix-egrep -regex ".*\.(pt|pth|ckpt|safetensors)$" -mtime +30
```

and when you feel it's safe to delete, only then add `rm`

```
find /mypath/ -regextype posix-egrep -regex ".*\.(pt|pth|ckpt|safetensors)$" -mtime +30 -exec rm
{} +
```

Resources

du alternatives

`du` is a powerful Unix tool, but it can be slow and usually requires additional postprocessing to sort the outcome to make the latter useful, for example I often use this formula `du -ahd1 | sort -rh` for getting disk usage of a single level sorted by size.

- dust - written in Rust, it runs faster than `du` and has a built in sorting. I find the no flags output to be confusing to read as it mixes many levels together and sub-sorts them, but it might be OK to others. `dust -F` and `dust -D` appear to be very useful and formatted well.
- baobab - analyzes partitions quite quickly and provides a UI to navigate the results based on sub-folder size with easy copying of the paths useful when needing to delete a lot of old large dataset and checkpoint folders. I think this tool is only useful on a desktop since it requires GNOME env.

Inter-Node and Intra-Node Networking Hardware

Subsections:

- Communication Patterns
- Network Debug
- Network Benchmarks

Introduction

It's not enough to buy/rent expensive accelerators to train and infer models fast. You need to ensure that your storage IO, CPU and network are fast enough to "feed the accelerator furnace". If this is not ensured then the expensive accelerators will be underutilized leading to lost $$, slower training time and inference throughput. While it can be any other of the mentioned components, the network is often the bottleneck during the training (assume your DataLoader is fast).

If your model fits on a single accelerator, you have little to worry about. But nowadays most models require several accelerators to load and LLM/VLM models require multiple compute nodes for training and some even for inference.

Most compute nodes contain 8 accelerators, some 4, others 16, and even more accelerators and recently there are some that have one super-accelerator per node.

When the model spans several accelerators and doesn't leave a single node all you need to worry about is fast Intra-node networking. As soon as the model requires several nodes, which is often the case for training as one can use multiple replicas to parallelize and speed up the training, then fast Inter-node networking becomes the key.

This article covers both types of networking hardware, reports their theoretical and effective bandwidths and explains how they inter-play with each other.

Glossary and concepts

You can safely ignore the many concepts and abbreviations listed here until you need them and then return here.

- ALU: Arithmetic Logic Units
- C2C: chip-to-chip, as in NVLink-C2C
- CPO: co-packaged optics - the optical engine sits inside the switch package instead of a pluggable transceiver
- DMA: Direct Memory Access
- EDR, HDR, NDR, XDR, GDR, LDR: InfiniBand per-lane data-rate generations - see InfiniBand
- EFA: Elastic Fabric Adapter
- GA: Generally Available - the product can actually be bought or rented, as opposed to announced, sampling, or spec'd only
- GDR: an InfiniBand per-lane data-rate generation - a roadmap target at 400Gbps per lane, as in the `InfiniBand GDR3200` row of the inter-node table - see InfiniBand. Beware that NCCL uses the same three letters for GPUDirect RDMA, the direct path between a NIC and accelerator memory, as in `NCCL_NET_GDR_LEVEL` - unrelated to this entry, and the clash is inherited from the two ecosystems that coined them, IBTA for the link generation and NVIDIA for the data path
- HCA: Host Channel Adapter
- HPC: High-performance Computing
- IB: InfiniBand
- IBTA: InfiniBand Trade Association
- ICI: Inter-Chip Interconnect - Google's scale-up fabric between TPU chips
- MFU: Model Flops Utilization (e.g. `mfu=0.5` at half-precision on A100 comes from getting 156TFLOPS,

because peak half-precision spec is 312TFLOPS, and thus `156/312=0.5`)

- NCCL: NVIDIA Collective Communications Library
- NIC: Network Interface Card
- NVL72: an NVLink domain of 72 accelerators; likewise NVL8 and NVL36
- NVLS: NVLink SHARP - the NCCL algorithm that reduces inside the NVSwitch
- NVLSTree: the NCCL algorithm that combines `NVLS` inside each node with a tree across nodes; multi-node only
- OPA: Omni-Path Architecture
- OPX: Omni-Path Express
- OSFP: Octal Small Form Factor Pluggable (transceiver)
- P2P: peer-to-peer - traffic between exactly two accelerators, as opposed to all-to-all
- QSFP-DD: Quad Small Form Factor Pluggable Double Density (transceiver)
- RC: Reliable Connection - the default connection type in verbs-based tools, connection-oriented with reliable in-order delivery
- RDMA: Remote Direct Memory Access
- RMA: Remote Memory Access - libfabric's name for one-sided read/write operations, the same class of operation as an RDMA write
- RoCE: RDMA over Converged Ethernet
- SHARP: Scalable Hierarchical Aggregation Reduction Protocol
- SRD: Scalable Reliable Datagram - AWS EFA's transport, which delivers reliably but does not preserve message order, unlike RC
- SuperNIC: NVIDIA's name for its recent high-rate network adapters
- TCPX, TCPXO: Google Cloud's GPUDirect transports for A3 machine types
- UEC: Ultra Ethernet Consortium - the body behind the Ultra Ethernet standard that AI NICs target
- VPC: Virtual Private Cloud - the general-purpose cloud network, as opposed to the accelerator fabric
- xGMI: Socket to Socket Global Memory Interface
- XGS: the cross-datacenter variant of Spectrum-X

Speed-related:

- Adaptive Routing: improves Static routing to enable out of order packets on the network. Packets are load balanced at each switch to better distribute the network workload.
- algbw: algorithm bandwidth - payload size divided by elapsed time
- Bi-directional, Duplex: a transmission from one point to another in both directions A <-> B, typically 2x the one-direction (unidirectional) rate
- Bisection Width: minimum number of links cut to divide the network into two parts (not necessarily equal). The bandwidth of those links is known as Bisection Bandwidth - which is often used as a metric for real network bandwidth). Sometimes it's referred to as the worst-case network capacity. Here is a good answer that explains this and related concepts, but it's unlikely you need to understand this other than knowing what is being meant, as chances are your cluster's topology has already been done by the provider.
- busbw: bus bandwidth - `algbw` scaled by a per-collective correction factor so it reflects the hardware bottleneck rather than the rank count
- GBps, GB/s: Gigabytes per secs (1GBps = 8Gbps) transferred in a channel
- Gbps, Gb/s: Gigabits per secs (1Gbps = 1/8GBps) transferred in a channel
- GT/s: GigaTransfers per second - the number of operations transferring data that occur in each second.
- Unidirectional: a transmission from one point to another in one direction A -> B

footnote: In the following sections pay close attention that 1GBps = 8Gbps.

Unidirectional vs Bidirectional (Duplex)

Most benchmarking / bandwidth measurement tools will report a unidirectional bandwidth. So be careful when you look

at unidirectional vs. bidirectional (duplex) speeds. Typically the latter is ~2x faster.

If you measure the bandwidth on your setup and it's about 40% of the advertised speed, carefully check if the advertised speed said duplex and if so half that and then your measured bandwidth should now be about 80% which is expected.

case study: for a while I couldn't understand why when I run the nccl-tests all_reduce benchmark on an A100 node with advertised 600GBps intra-node speed I was getting only 235GBps (40%) until Horace He kindly pointed out that I should be looking at unidirectional speed which is 300GBps, and then I get 80% of the theoretical spec which checks out.

That ~80% expectation holds on newer hardware too. On a node of 8x H200 (NVLink 4, NVSwitch-connected) a pairwise copy measures 396.40GBps unidirectional against the 450GBps spec, or 88%, and 783.73GBps duplex against 900GBps, or 87% - so on this fabric running both directions at once costs almost nothing, measured with nvbandwidth:

```
./nvbandwidth -i 10 -t device_to_device_memcpy_write_ce                # 396.40 GB/s per pair
./nvbandwidth -i 10 -t device_to_device_bidirectional_memcpy_write_ce  # 783.73 GB/s per pair,
total
```

footnote: the bidirectional testcase prints `Write1`, `Write2` and a combined `total` matrix - read the total rather than doubling one direction.

A pairwise copy is the right instrument for this comparison because it moves bytes and nothing else. A *collective* on the same hardware can report more than the unidirectional spec, which is a different effect entirely - see SHARP.

Cluster networks

Each node of the cluster has 3 networks, each running at a very different speed from each other.

1. Frontend
2. Backend
3. Out-of-band

Frontend networking

Frontend networking is typically for the Internet connection (e.g. downloading python packages and offloading to the cloud storage), distributed network storage (e.g. checkpoints and datasets) and orchestration (e.g. SLURM and Kubernetes). As of 2026-08 a typical node is likely to have a single 100-400Gbps connection.

footnote: not all clusters will have external Internet connection available, e.g. many HPC environments only provide external access via special cpu-only nodes.

Backend networking

Backend networking is to perform GPU-to-GPU connectivity which allows training and inference to scale to multiple accelerators (e.g. all-reduce, all-gather and other collective comms). This is the most important part of the AI cluster. Typically this would be either an InfiniBand or RoCEv2 Ethernet. It then breaks down into intra-node networking and inter-node networking - the GPUs on the same node typically can communicate with each other at faster speed than with GPUs on other nodes. Here the typical top unidirectional speeds as of 2026-08 would be around 900GBps per accelerator for intra-node (NVLink 5, as on B200/B300) and 100GBps per accelerator, or 800GBps per node, for inter-node (8x ConnectX-8, as on DGX B300 XDR) - so about an order of magnitude apart per accelerator, a ratio that has held for several generations because both sides keep doubling together. That is the gap between the links; the gap you actually measure with a collective is much smaller, for reasons explained in Inter-node speed depends on intra-node speed. There will be at least one backend connection per accelerator and at times there can be multiple connections per accelerator, especially if low bandwidth NICs are used.

footnote: not all providers will match the industry's standard networking speeds - on some the inter-node networking speed could be up to 10x slower. So always check what you get.

Out-Of-Band networking

Out-Of-Band (OOB) networking is used for bootstrapping backend networking, monitoring node's health, remote re-imaging of the nodes, etc. It typically uses a single slow 1Gbps ethernet connection.

RDMA networking

Remote Direct Memory Access is like DMA (Direct Memory Access) on the node, but across nodes. It allows data exchange between nodes w/o the overhead using the local processor, OS kernel and caches, which is what TCP/IP uses. The 3 main implementations are:

1. InfiniBand
2. RDMA over Converged Ethernet (RoCE) (IB or UDP-based RDMA)
3. iWARP (TCP-based RDMA)

Here is a good overview article.

Intra-node networking

This is also known as scale-up networking.

There are multiple platforms/solutions out there that provide intra-node networking:

1. Generic: PCIe
2. NVIDIA: NVLink and NVSwitch
3. AMD: Infinity Fabric
4. Intel: Gaudi2, Gaudi3
5. AWS: NeuronLink (Trainium)
6. Google: ICI (TPU)
7. Huawei: UB Link (Ascend)

All-to-all bandwidth

Here is intra-node unidirectional theoretical all-to-all peak bandwidth cross-comparison for current solutions sorted by bandwidth:

Generally available:

Interconnect	Accelerator	GBps	Notes
NVIDIA NVLink 5	B200, B*	900.0	1
AWS NeuronLink 3	Trainium2	640.0	2
Intel	Gaudi3	600.0	
Google ICI	TPU7x	600.0	4
AMD XGMI	MI355X	535.5	
AMD XGMI	MI350X	535.5	
NVIDIA NVLink 4	H100, H*	450.0	1

Interconnect	Accelerator	GBps	Notes
AMD XGMI	MI325X	448.0	
AMD XGMI	MI300X	448.0	
AMD XGMI	MI250X	350.0	
NVIDIA NVLink 3	A100	300.0	1
Intel	Gaudi2	300.0	
PCIe 5		63.0	
PCIe 4		31.0	

Announced, availability not confirmed:

Interconnect	Accelerator	GBps	Notes
Huawei UB Link	Ascend 950DT	840.0	5
NVIDIA NVLink 6	Rubin	1800.0	1
PCIe 6		121.0	3

Notes:

1. NVSwitch operates at the same speed as NVLink of that generation. See NVSwitch.
2. AWS publishes `NeuronLink-v3 ... 1.28 TB/sec bandwidth per chip` for Trainium2 without declaring directionality, so it's halved here per the directionality note below. AWS doesn't publish a per-link rate or a link count, so Trainium2 can't be placed in the peer-to-peer table below.
3. PCIe 6 is listed under `Announced, availability not confirmed` because no shipping accelerator attaches at Gen6 as of 2026-07-31 - current parts are Gen5 x16 (NVIDIA H200 SXM lists `PCIe Gen5`, AMD MI350X/MI355X list `PCIe 5.0 x16`), so 63GBps remains today's ceiling for accelerator-to-accelerator PCIe traffic. Gen6 already ships on the NIC side - NVIDIA markets ConnectX-8 as bringing "PCIe Gen6 connectivity in a single device", which is what lets one adapter feed 800Gbps.
4. Google publishes a "Bidirectional inter-chip interconnect (ICI) bandwidth per chip (GBps)" of 1200 for TPU7x, halved here per the note above. Peer-to-peer is the per-axis figure, "bi-directional bandwidth of 200 GBps per axis" - and 6 neighbors in the 3D torus at 200 each is exactly the 1200 total, so the two figures corroborate. `GA` is `?` because Google documents TPU7x fully without stating an availability stage. See TPU7x.
5. Huawei publishes a per-cabinet total interconnect bandwidth of up to 64 x 1.68TBps bidirectional for the Atlas 950 SuperPoD - 1.68TBps bidirectional per accelerator, halved here. Only the Chinese pages carry it; the English ones state no per-NPU figure. Peer-to-peer is unknown, so there is no row in the peer-to-peer table. `GA` is `?` because Huawei publishes no availability for Ascend 950DT anywhere reachable - its own English site serves a plain downloader only a navigation shell, in which `950DT` does not appear at all. China-only distribution is not the reason: once availability is confirmed the row belongs with the available hardware, annotated as China-only. See UB Link.

General notes:

- Pay close attention to when the spec says unidirectional vs bidirectional (duplex) speeds - if you read an online spec and it doesn't explicitly declare the directionality - look for an answer. I had to research many

docs to figure it out in some of the tables below as some vendors omit this crucial information in the published specs. I even had to edit a few wiki pages to add the missing information. Remember that for the vendors the bigger, the better so almost always they will use the duplex number, which is typically 2x bigger than the unidirectional one.

Peer-to-peer bandwidth

Some vendors have their all-to-all and peer-to-peer (GPU-to-GPU) bandwidth the same, while others don't. For example, AMD MI300X/MI325X are 64GBps GPU-to-GPU (peer-to-peer), but 448GBps in total on a board of 8 accelerators, since `64*7=448`. Likewise MI350X/MI355X are 76.5GBps peer-to-peer and 535.5GBps all-to-all, since `76.5*7=535.5`.

Here is the intra-node unidirectional theoretical peer-to-peer peak bandwidth cross-comparison for current solutions sorted by bandwidth:

Generally available:

Interconnect	Accelerator	GBps
NVIDIA NVLink 5	B200, B*	900.0
Intel	Gaudi3	600.0
NVIDIA NVLink 4	H100, H*	450.0
NVIDIA NVLink 3	A100	300.0
Intel	Gaudi2	300.0
Google ICI	TPU7x	100.0
AMD XGMI	MI355X	76.5
AMD XGMI	MI350X	76.5
AMD XGMI	MI325X	64.0
AMD XGMI	MI300X	64.0
PCIe 5		63.0
AMD XGMI	MI250X	50.0
PCIe 4		31.0

Announced, availability not confirmed:

Interconnect	Accelerator	GBps
NVIDIA NVLink 6	Rubin	1800.0
PCIe 6		121.0

note: PCIe carries the same number in both tables, but for the opposite reason to NVLink. NVLink matches because NVSwitch lets a single pair light up every link - you get the whole fabric either way. PCIe matches because an accelerator has just one x16 host link, so that link is the ceiling whether it talks to one peer or to seven. And unlike NVLink or Infinity Fabric, where every pair has the same fabric path, the PCIe number depends on where the two devices sit - under the same

switch, across the root complex, or across CPU sockets are all different paths (`nvidia-smi topo -m` tells you which). So treat the PCIe rows as a best case for a well-placed pair, not as a figure that holds for any two devices in the node.

note: the PCIe rows assume an x16 attachment, which is what current accelerators use - NVIDIA H200 SXM lists `PCIe Gen5` at 128GBps duplex, AMD MI350X/MI355X list `PCIe 5.0 x16`, and A100 is Gen4 x16. If a part is attached at x8, halve the figure. Note also that as of 2026-07-31 no shipping accelerator attaches at Gen6 - so 63GBps is today's real ceiling for accelerator-to-accelerator PCIe traffic, and the PCIe 6 row is there for the platforms that will. Gen6 *is* already shipping elsewhere in the node: NVIDIA markets ConnectX-8 as bringing "PCIe Gen6 connectivity in a single device", which is what lets one adapter feed 800Gbps (100GBps) since Gen5 x16 tops out at 63GBps.

When peer-to-peer bandwidth is much lower than all-to-all it means that if you don't use all of the accelerators on the node by the same application, you will end up with a much lower bandwidth and your application will have a performance impact if the accelerators have to communicate between each others.

To validate this the all_reduce_bench.py was run on a 8x GPU AMD MI300X node with a 4GiB payload and the `busbw` measurements were:

- 2 GPUs: 47.671GBps
- 8 GPUs: 312.912GBps

i.e. 2 GPUs performed 6.5x slower than 8.

So if you have you to deploy TP=2, TP=4, or ZeRO-DP/FSDP over 2 or 4 GPUs, be it training or inference, the network will become a bottleneck. If you use TP=1 or TP=8 or ZeRO-DP/FSDP over 8 GPUs, or DP over 1-GPU replicas there is no problem. (If you're not sure what TP/ZeRO-DP/DP mean please see model-parallelism.)

You will find the details analysis of each technology in the following sections.

PCIe

PCIe is a high-speed serial computer expansion bus standard that can be found even on the cheapest computer desktop.

Generally available:

Interconnect	Lane/ Direction (GBps)	Lanes	Uni-dir. (GBps)	Duplex (GBps)
PCIe 4	~2.0	16	31	62
PCIe 5	~4.0	16	63	126
PCIe 6	~7.5	16	121	242

Announced, availability not confirmed:

Interconnect	Lane/ Direction (GBps)	Lanes	Uni-dir. (GBps)	Duplex (GBps)
PCIe 7	~15.0	16	242	484
PCIe 8	~30.0	16	484	968

If one compares the latest generations of different intra-node networking technologies (see the following sections) PCIe is

usually an order of magnitude behind.

The table above is spec. What a PCIe link actually delivers is a separate question, and one nobody seems to publish - so here is a measurement. On a node of 8x H200 where each accelerator has its own PCIe 5 x16 link, nvbandwidth reports 55.43GBps host-to-device and 55.06GBps device-to-host, which is 88% and 87% of the 63GBps row above. The eight links came out within 0.10GBps of each other, with `nvbandwidth` reporting a coefficient of variation of 0.00.

To reproduce, first confirm the generation and width, since the 63GBps figure only applies to PCIe 5 at x16:

```
nvidia-smi --query-gpu=index,pcie.link.gen.current,pcie.link.width.current --format=csv
```

then, with nvbandwidth:

```
./nvbandwidth -i 10 -t host_to_device_memcpy_ce
./nvbandwidth -i 10 -t device_to_host_memcpy_ce
```

caveat: `host_to_device_memcpy_ce` measures whatever the host-to-device path happens to be on that platform, so read it against the fabric the machine actually uses. On an x86 host with PCIe-attached SXM accelerators that path is PCIe, which is what the 88% above refers to. On a Grace-Blackwell system the very same testcase measures NVLink-C2C instead and reports several hundred GBps - same command, different fabric, an order of magnitude apart.

footnote: a released specification is not shipping silicon. As of 2026-07-30 PCI-SIG lists `PCI Express Base Specification Revision 7.0` as released, while PCIe 8.0 exists only as a members-only draft, targeted for 2028. And even a finished spec takes years to reach products - PCIe 6.0 hardware didn't launch until August 2025, about three years after its spec was done. So read the last rows as where the standard is heading, not as what you can buy.

NVLink

footnote: `nvidia-smi nvlink -s` reports the raw per-link signalling rate, and summing it overshoots the advertised aggregate. A B200 lists 18 links at 53.125GBps = 956.25GBps against the advertised 900GBps, and an H100 lists 18 at 26.5625GBps = 478.125GBps against 450GBps. Both land at a ratio of 0.9412, which is the encoding overhead, so multiply the sum by ~0.94 to get back to the spec figure.

- NVLink is a wire-based serial multi-lane near-range communications link developed by NVIDIA. Here is the What Is NVLink blog post with more background on it.

I found the NVLink wiki page to be quite difficult to follow, so I will try to help bring clarity into this. And I'm pretty sure as of 2026-08 some of the numbers on that wiki page are bogus and it doesn't look like NVIDIA maintains that page.

Effective payload rate of intra-node GPU-to-GPU communication hardware:

Generally available:

Interconnect	Lane/ Direction (GBps)	Lanes	Links	Uni-dir. (GBps)	Duplex (GBps)	GPU
NVLink 1	2.50	8	4	80	160	P100
NVLink 2	3.125	8	6	150	300	V100
NVLink 3	6.25	4	12	300	600	A100
NVLink 4	12.50	2	18	450	900	H100, H200, GH200

Interconnect	Lane/ Direction (GBps)	Lanes	Links	Uni-dir. (GBps)	Duplex (GBps)	GPU
NVLink 5	25.00	2	18	900	1800	B200, B*, GB*

Announced, availability not confirmed:

Interconnect	Lane/ Direction (GBps)	Lanes	Links	Uni-dir. (GBps)	Duplex (GBps)	GPU
NVLink 6	25.00	2	36	1800	3600	Rubin

There is a good overview of evolution of NVLink (1 to 4) here.

The largest PCIe 16x slot has 16 lanes. Smaller slots have less lanes, 1x == 1 lane.

NVIDIA Rubin nodes come equipped with PCIe 6 and NVLink 6. So there NVLink is ~15x faster than PCIe.

NVIDIA Blackwell nodes come equipped with PCIe 6 and NVLink 5. So there NVLink is ~7x faster than PCIe.

NVIDIA Hopper nodes typically come equipped with PCIe 5 and NVLink 4. So there NVLink is ~7x faster than PCIe.

Let's look at several examples of nodes and correlate the theory with reality.

If you use multiple GPUs the way cards are inter-connected can have a huge impact on the total training time. If the GPUs are on the same physical node, you can run:

```
nvidia-smi topo -m
```

and it will tell you how the GPUs are inter-connected.

On a machine with dual-GPU and which are connected with NVLink, you will most likely see something like:

```
        GPU0    GPU1    CPU Affinity    NUMA Affinity
GPU0     X      NV2     0-23            N/A
GPU1    NV2      X      0-23            N/A
```

on a different machine w/o NVLink you may see:

```
        GPU0    GPU1    CPU Affinity    NUMA Affinity
GPU0     X      PHB     0-11            N/A
GPU1    PHB      X      0-11            N/A
```

The report includes this legend:

```
  X    = Self
```

```
  SYS  = Connection traversing PCIe as well as the SMP interconnect between NUMA nodes (e.g.,
QPI/UPI)
  NODE = Connection traversing PCIe as well as the interconnect between PCIe Host Bridges within
a NUMA node
  PHB  = Connection traversing PCIe as well as a PCIe Host Bridge (typically the CPU)
  PXB  = Connection traversing multiple PCIe bridges (without traversing the PCIe Host Bridge)
  PIX  = Connection traversing at most a single PCIe bridge
  NV#  = Connection traversing a bonded set of # NVLinks
```

So the first report `NV2` tells us the GPUs are interconnected with 2 NVLinks, and the second report `PHB` shows we have a typical consumer-level PCIe+Bridge setup.

Check what type of connectivity you have on your setup. Some of these will make the communication between cards faster (e.g. NVLink), others slower (e.g. PHB).

Depending on the type of scalability solution used, the connectivity speed could have a major or a minor impact. If the GPUs need to sync rarely, as in DDP, the impact of a slower connection will be less significant. If the GPUs need to send messages to each other often, as in ZeRO-DP, then faster connectivity becomes super important to achieve faster training.

Now, let's look at the topology of the A100 and H100 nodes:

- A100 topology:

```
$ nvidia-smi topo -m
      GPU0  GPU1  GPU2  GPU3  GPU4  GPU5  GPU6  GPU7  CPU Affinity  NUMA Affinity
GPU0   X    NV12  NV12  NV12  NV12  NV12  NV12  NV12   0-23         0
GPU1  NV12   X    NV12  NV12  NV12  NV12  NV12  NV12   0-23         0
GPU2  NV12  NV12   X    NV12  NV12  NV12  NV12  NV12   0-23         0
GPU3  NV12  NV12  NV12   X    NV12  NV12  NV12  NV12   0-23         0
GPU4  NV12  NV12  NV12  NV12   X    NV12  NV12  NV12  24-47         1
GPU5  NV12  NV12  NV12  NV12  NV12   X    NV12  NV12  24-47         1
GPU6  NV12  NV12  NV12  NV12  NV12  NV12   X    NV12  24-47         1
GPU7  NV12  NV12  NV12  NV12  NV12  NV12  NV12   X    24-47         1
```

You can see there are 12 NVLinks and 2 NUMA Groups (2 CPUs w/ 24 cores each)

- H100 topology:

```
$ nvidia-smi topo -m
      GPU0  GPU1  GPU2  GPU3  GPU4  GPU5  GPU6  GPU7  CPU Affinity  NUMA Affinity
GPU0   X    NV18  NV18  NV18  NV18  NV18  NV18  NV18   0-51         0
GPU1  NV18   X    NV18  NV18  NV18  NV18  NV18  NV18   0-51         0
GPU2  NV18  NV18   X    NV18  NV18  NV18  NV18  NV18   0-51         0
GPU3  NV18  NV18  NV18   X    NV18  NV18  NV18  NV18   0-51         0
GPU4  NV18  NV18  NV18  NV18   X    NV18  NV18  NV18  52-103        1
GPU5  NV18  NV18  NV18  NV18  NV18   X    NV18  NV18  52-103        1
GPU6  NV18  NV18  NV18  NV18  NV18  NV18   X    NV18  52-103        1
GPU7  NV18  NV18  NV18  NV18  NV18  NV18  NV18   X    52-103        1
```

You can see there are 18 NVLinks and 2 NUMA Groups (2 CPUs w/ 52 cores each)

Of course, other A100 and H100s node reports may vary, e.g. the number of cpu cores is likely to be different.

NVLink-C2C

This is a high-bandwidth connection between Grace CPU and GPUs on GH200 and GB200+ modules, Vera CPU and Rubin GPUs.

As of 2026-08 there is no public spec of the speed, but I found 450GBps unidirectional mentioned here for GB200. As compared to 900GBps unidirectional bandwidth for NVLink-5 - so half the speed of the latter.

request: I'm looking for an official spec if you find one please let me know.

Generally available:

Architecture	Uni-dir. (GBps)	NVLink (GBps)	NVLink gen
Grace/Hopper	450	900	4
Grace/Blackwell	450	900	5

Announced, availability not confirmed:

Architecture	Uni-dir. (GBps)	NVLink (GBps)	NVLink gen
Vera/Rubin	900	1800	6

Next, it's important to understand that these speeds are of a standalone C2C technology and it can be much lower when integrated into the system, when bottlenecked by other components.

On DGX Station (comprised of half the GB300 module) I benchmarked ~80% unidirection and ~38% duplex efficiency vs theoretical bandwidth using the nvbandwidth benchmark:

```
$ ./nvbandwidth
[...]
Running host_to_device_memcpy_ce.
memcpy CE CPU(row) -> GPU(column) bandwidth (GB/s)
           0
 0    359.05

SUM host_to_device_memcpy_ce 359.05

Running device_to_host_memcpy_ce.
memcpy CE CPU(row) <- GPU(column) bandwidth (GB/s)
           0
 0    377.15

SUM device_to_host_memcpy_ce 377.15

Running host_to_device_bidirectional_memcpy_ce.
```

```
memcpy CE CPU(row) <-> GPU(column) bandwidth (GB/s)
           0
 0    175.38

SUM host_to_device_bidirectional_memcpy_ce 175.38

Running device_to_host_bidirectional_memcpy_ce.
memcpy CE CPU(row) <-> GPU(column) bandwidth (GB/s)
           0
 0    171.76
 [...]
```

Here is the math (I averaged two direct bandwidth reports as the reports aren't the same for each direction)

```
(359+377)/2 / 450 = 0.82
(175+171)/2*2 / 900 = 0.38
```

Essentially, the end-to-end doesn't support the duplex C2C spec, but delivers half the unidirectional speed when communicated in both directions. The wire is duplex, but LPDDR5X memory bandwidth on Grace prevents it from running at full speed in both directions, as the bidirectional test makes both directions share that same memory bandwidth. It's probably not a problem in most current ML applications that normally would send data in one direction H2D / D2H at the same time, but if real duplex is needed beware of this issue.

NVSwitch

footnote: on a rack-scale NVLink system this makes `nvidia-smi topo -m` unusually informative - it reports `NV#` connections to accelerators in other chassis, because those are inside the same scale-up domain. On a conventional node the same tool tells you nothing about inter-node connectivity, since it only knows about NVIDIA's own fabrics.

NVSwitch can connect more than 8 GPUs at the speed of NVLink. It's advertised to connect up to 256 GPUs in the future generations of the switch.

The benefit of connecting more than 8 GPUs at the speed of NVLink is that it allows all-to-all GPU communications at a much faster speed than any intra-node hardware can provide. And with ever increasing compute speeds the network is the likely bottleneck leading to underutilized super-expensive GPUs.

For example, in the universe of Tensor Parallelism (Megatron), one doesn't use TP degree of more than 8, because TP is only efficient at NVLink speed. ZeRO-DP (DeepSpeed/FSDP) would also run much faster if the whole cluster uses NVLink speed and involves no slow inter-node connections.

NVSwitch is used for intra-node connectivity.

NVSwitch gen 1 came out with V100, gen 2 with A100, gen 3 with H100, and gen 4 with B200 - the speed corresponds to the NVLink version of the same technology.

The NVIDIA DGX H100 connects eight H100 GPUs through four third-generation NVSwitch chips. Each H100 has 900GBps of bidirectional NVLink 4 bandwidth, which NVIDIA still lists on a live page in its NVLink specifications table under `Fourth Generation`. NVIDIA separately specifies 3.6TBps of system bisection bandwidth and 450GBps of reduction bandwidth. These describe different scopes: 900GBps is the interface bandwidth available to one GPU, while 3.6TBps is aggregate traffic across a balanced partition of the eight-GPU fabric; the number of NVSwitch chips doesn't multiply the bandwidth of one GPU.

NVIDIA DGX A100 has 6 switches of 12 NVLinks for a total of 72.

A DGX H100 SuperPOD scalable unit contains 32 DGX H100 systems (256 GPUs), but each DGX remains its own in-node NVSwitch domain. The systems scale out over a separate Quantum-2 NDR400 InfiniBand compute fabric, so cross-node bandwidth must be described from the ConnectX-7/InfiniBand topology rather than by halving or summing the internal NVLinks. See InfiniBand.

Additionally, NVSwitch gen3 and higher comes with NVIDIA Scalable Hierarchical Aggregation Reduction Protocol (SHARP) which can boost both the intra- and inter-node speeds for `all-reduce`. NCCL versions now have `NCCL_ALGO=NVLS` which boosts the intra-node `all-reduce` bandwidth up to 30%, and inter-node `all-reduce` by about 25%.

Recently GB200 NVL72 has been introduced, which uses NVSwitch to put 72 Blackwell GPUs into a single node all inter-connected at NVLink 5 900GBps unidirectional speed. So instead of having a 8-gpu node, now we have a 72-gpu node (even though physically they don't all reside on the same board).

Infinity Fabric / xGMI

AMD MI* Accelerators intra-node communication is performed by AMD Infinity Fabric, which is also known as xGMI (Socket to Socket Global Memory Interface).

This is AMD's answer to NVLink.

The following is the all-to-all bandwidth.

Generally available:

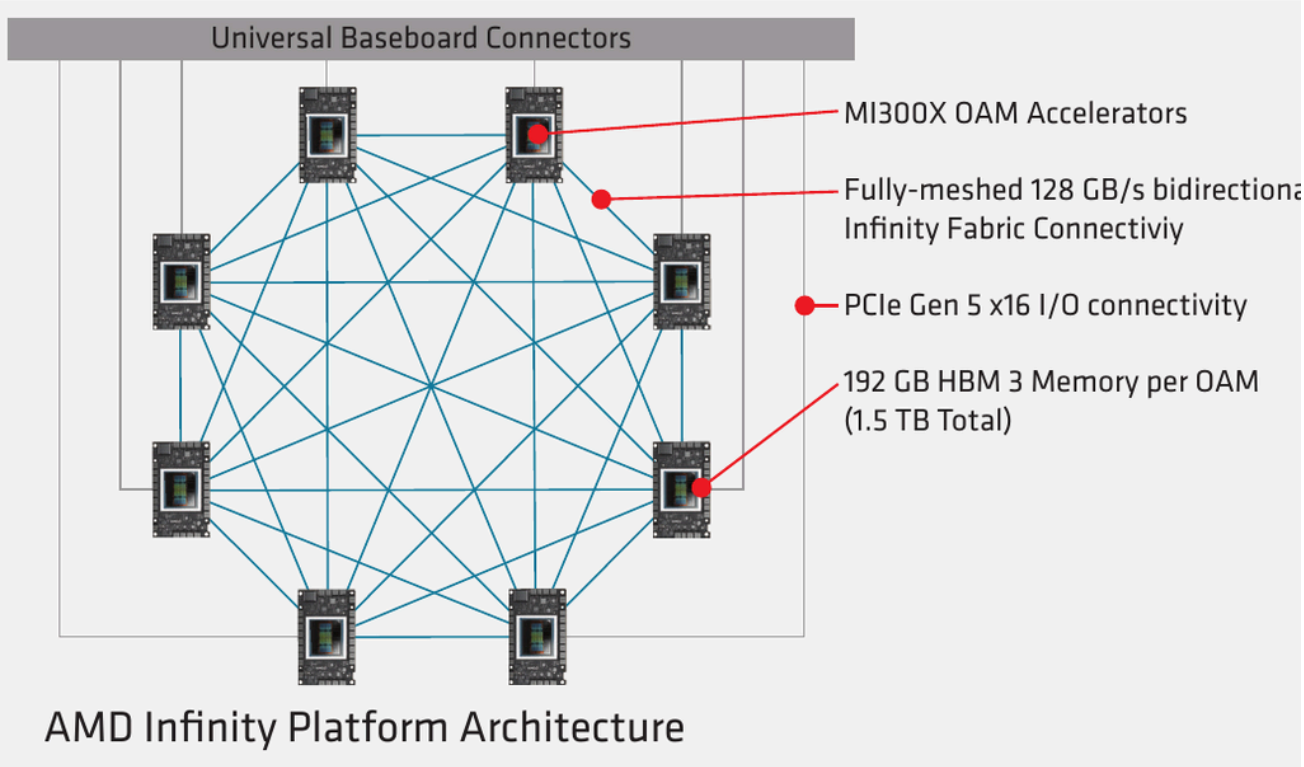
Interconnect	Link/ Direction P2P (GBps)	Links	Uni-dir. all-to-all (GBps)	Duplex all-to-all (GBps)
MI355X	76.5	7	535.5	1071
MI350X	76.5	7	535.5	1071
MI325X	64	7	448	896
MI300X	64	7	448	896
MI250X	50	7	350	700

Announced, availability not confirmed:

Interconnect	Link/ Direction P2P (GBps)	Links	Uni-dir. all-to-all (GBps)	Duplex all-to-all (GBps)
MI455X	??	??	1800	3600

The peer-to-peer bandwidth is just that of a single link/direction (the 2nd column). This means that unless you use the whole 8-GPU node in a single process group you will have a 7x slower comms performance. See Peer-to-peer bandwidth for details.

Other intra-node solutions typically have the same all-to-all and peer-to-peer intra-node bandwidth, so Infinity Fabric appears to be dramatically slower, unless the workload either doesn't use intra-node comms or the whole node is engaged in a single workload.



Platform specs:

- MI250X
- MI300x
- MI325X
- MI350X
- MI355X

footnote: the MI455X row is `??` where AMD publishes nothing. For MI3* AMD gives both `Infinity Fabric Links` and `Peak Infinity Fabric Link Bandwidth`, which is what lets the per-link and all-to-all columns be filled and the 7x peer-to-peer penalty be shown. For MI455X it publishes only `Scale-up Bandwidth per GPU: 3.6 TB/s` - no link count, no per-link rate - so the all-to-all columns carry that figure halved per the convention below, and the peer-to-peer columns can't be filled at all. Whether MI4* still divides its bandwidth per peer the way MI3* does, or switches to a fabric where a single pair can use all of it, is exactly what those two `??` would answer. AMD also never uses the words "all-to-all" or "any-to-any" about it, so treat 1800 as the accelerator's total scale-up bandwidth rather than a confirmed all-to-all figure.

footnote: AMD publishes a single `Peak Infinity Fabric Link Bandwidth` per link without saying whether it is uni- or bi-directional - 153GBps for MI350X/MI355X and 128GBps for MI300X/MI325X. Vendors quote the larger number, so these are read as duplex and halved for the per-direction column above. AMD also lists an 8th, scale-out link per GPU (128GBps peak), which is not part of the intra-node all-to-all figures.

Gaudi2

According to Gaudi2 spec, these nodes provide the same 100GbE RoCE v2 RDMA hardware for inter- and intra-node connectivity (24x 100Gbps per card).

- intra-node: 8x 7x3 NICs - 300Gbps card to card
- inter-node: 8x 1x3 NICS - for a total of 2.4Tbps (300GBps)

Gaudi3

According to Gaudi3 spec, these nodes provide the same setup as Gaudi2 except the cards are 2x faster using 200GbE RoCE v2 RDMA for inter- and intra-node connectivity (24x 200Gbps per card).

- intra-node: 8x 7x3 NICs - 600Gbps card to card
- inter-node: 8x 1x3 NICS - for a total of 4.8Tbps (600GBps)

NeuronLink v3

NeuronLink v3 (spec) is the intra-node equivalent of NVLink for AWS Trainium2, but it's a point-to-point architecture, like AMD MI* so it can't take advantage of the other Trainium2 chips' NeuronLink v3 unless they are engaged in the same process group. This technology is based on PCIe-5.0 (so 32Gbps per lane unidirectional).

NeuroLink v3 also has an inter-node use in addition to EFA v3.

Number of Trainium2 chips per node and intra-node network speeds:

- Trainium2: 16 chips interconnected at 128GBps peer-to-peer undirectional (32 PCIe lanes) and each Trainium2 connects to 3 other chips
- Trainium2 Ultra: 64 chips - the 16 chip groups are the same as non-Ultra, plus these 4 groups are interconnected at 64GBps with each other.

Like TPU it is used in a 3D Torus structure. Here different axis connect at different speeds, so the total all-to-all bandwidth per chip is 640GBps unidirectional (`128GBps * 4 intra-node neighbors + 64GBps * 2 inter-node neighbors`)

When their spec suggests 1024GBps/chip intra-instance bandwidth, it is bidirectional, so only 512GBps/chip unidirectional - and it comes from `128GBps * 4 intra-node neighbors` (and only if all 4 chips are engaged).

Inter-node networking

This is also known as scale-out networking.

As inter-node hardware used to be about of an order of magnitude slower than intra-node hardware in this universe Gbps are used instead of GBps. (1GBps = 8Gbps) (The links themselves are still about an order of magnitude apart, but the practical gap for collectives is much smaller, because an inter-node collective also does most of its work over the intra-node links - see Inter-node speed depends on intra-node speed)

When it comes to inter-node networking hardware, there are the well established InfiniBand from NVIDIA and a few other players, various NVLink-based NVIDIA products and there are many new comers that mainly are coming from compute cloud providers who can't compete on the slim margin renting out someone else's hardware so they build their own (AWS EFA, GCP GPUDirect-TCPX), and there are also HPE and Cornelis Networks with recently updated products.

As of 2026-07-28, here is an inter-node unidirectional theoretical peak bandwidth cross-comparison for current platform configurations and interconnect generations. An interface here is a fabric-facing NIC or adapter port, not a SerDes lane or a switch port. Total bandwidth is `interfaces * rate / 8`. The `Shipped` column says whether you can actually get the hardware today (V) or whether it's still a roadmap or pre-GA target (X).

Sorted by Total unidirectional bandwidth descending, then Rate/interface descending, then Platform/example node ascending:

Generally available:

Platform/ example node	NICs per node	Rate/ interface (Gbps)	Total Uni-dir. (GBps)	Notes
NVIDIA DGX B300 XDR	8	800	800	1,19

Platform/ example node	NICs per node	Rate/ interface (Gbps)	Total Uni-dir. (GBps)	Notes
AWS EFA v4 (P6-B300)	16	400	800	2,20,31
Intel Gaudi3	24	200	600	3,21
NVIDIA DGX H100 NDR	8	400	400	4,22
Omni-Path CN5000 example	8	400	400	17,18
AWS EFA v4 (P6-B200)	8	400	400	2,30,31
AWS EFA v3 (P5en/Trn2)	16	200	400	2,23,31
AWS EFA v2 (P5/P5e)	32	100	400	2,23,31
Intel Gaudi2	24	100	300	5,21
InfiniBand XDR200	2	800	200	9,11
GCP A3 Mega TCPXO	8	200	200	6,24,29
GCP A3 High TCPX	4	200	100	6,25,29
HPE Slingshot example	4	200	100	7,26
Omni-Path CN100 example	8	100	100	8,27
InfiniBand NDR400	1	400	50	10,12
AWS EFA v1 (P4d)	4	100	50	2,28,31

Announced, availability not confirmed:

Platform/ example node	NICs per node	Rate/ interface (Gbps)	Total Uni-dir. (GBps)	Notes
Omni-Path CN6000 example	8	800	800	13,14
InfiniBand GDR3200	2	1600	400	15,16

Notes:

1. NVIDIA DGX B300 specifications
2. AWS EC2 accelerated-computing network specifications
3. Intel Gaudi3 white paper
4. NVIDIA DGX H100/H200 user guide
5. Intel Gaudi2 data sheet
6. Google Cloud GPUDirect-TCPX/TCPXO
7. HPE Slingshot 11 QuickSpecs
8. Cornelis CN-100HFA specifications
9. InfiniBand Trade Association XDR specification release
10. NVIDIA NDR cabling guide

11. Originally written as `8x200Gbps` lane arithmetic - eight 200Gbps XDR lanes are two full-width four-lane XDR ports, not a documented eight-port node. Legacy label from an earlier edition.
12. Originally written as `4x100Gbps` lane arithmetic - four 100Gbps NDR lanes are one full-width four-lane NDR port, not four NICs. Legacy label from an earlier edition.
13. Cornelis CN6000 product page and AMD MI400 reference architecture
14. 800Gbps product sampling in 2026; GA target Q4-2026; eight-adapter node remains illustrative.
15. InfiniBand Trade Association roadmap
16. Originally written as `8x400Gbps` lane arithmetic - eight 400Gbps GDR lanes are two full-width four-lane GDR ports. GDR is not a standardized shipping platform; the original roadmap target was 2025.
17. Cornelis CN5000 launch
18. 400Gbps family began shipping in June 2025 and broadly available from Q3 2025; eight-adapter node remains illustrative. The original roadmap target was Q2-2025.
19. Eight single-port ConnectX-8 adapters.
20. Sixteen fabric-facing EFA interfaces.
21. Integrated RoCEv2 interfaces.
22. Eight single-port ConnectX-7 adapters.
23. Common 3.2Tbps configuration.
24. Eight accelerator-fabric interfaces.
25. Four accelerator-fabric interfaces.
26. Illustrative four-interface node.
27. Illustrative eight-adapter node.
28. Four fabric-facing EFA interfaces.
29. Google publishes a higher headline number for these machine types - 1800Gbps max network bandwidth for A3 Mega and 1000Gbps for A3 High. Those totals include the VPC/frontend NIC; the figures above count only the accelerator fabric, so `8x200=1600Gbps` and `4x200=800Gbps` respectively. The ~200Gbps difference in each case is the general-purpose network interface, which doesn't carry collectives.
30. Eight fabric-facing EFA interfaces at 400Gbps each, device-reported via `/sys/class/infiniband/rdmap*/ports/1/rate`, for the 3.2Tbps AWS publishes per P6-B200 instance.
31. The `v1`-`v4` EFA generation labels are AWS's own, taken from the supported instance types tables, which group instances under `Nitro v3 (EFA v1)` through `Nitro v6 (EFA v4)` headings. The generation follows the Nitro version, not the accelerator - see EFA.

These are common/popular node setups - some custom nodes may have a different configuration more often with less NICs and rarely with more NICs. And, yes, AWS EFA v2 puts 32 NICs on each node - that must be a lot of wires.

footnote: Note how the once order-of-magnitude difference between inter- and intra-node bandwidth is starting to disappear - I have recently rescaled the speeds here from Gbps to GBps.

You will find the details analysis of each technology in the following sections.

Network adapters

The tables in this chapter mostly count node-level bandwidth, but you buy adapters. NVIDIA calls its recent ones SuperNICs, AMD's AI NIC line is Pensando. This is the per-adapter throughput, which is not the same thing as a per-port rate - most of these adapters carry more than one port and vendors rarely say how many. Sorted by throughput descending:

Generally available:

Adapter	Vendor	Protocol	Throughput per adapter (Gbps)	Notes
ConnectX-8 SuperNIC	NVIDIA	InfiniBand XDR, Ethernet	800	1,4

Adapter	Vendor	Protocol	Throughput per adapter (Gbps)	Notes
ConnectX-7 HCA	NVIDIA	InfiniBand NDR, Ethernet	400	1,4

Announced, availability not confirmed:

Adapter	Vendor	Protocol	Throughput per adapter (Gbps)	Notes
ConnectX-9 SuperNIC	NVIDIA	Ethernet	1600	1,7
CN6000 SuperNIC	Cornelis	Omni-Path, RoCEv2, Ultra Ethernet	800	8,9
Pensando Vulcano	AMD	Ultra Ethernet	800	2,5,6
Pensando Pollara 400	AMD	Ethernet	400	2,3,7

Notes:

1. NVIDIA InfiniBand adapters - NVIDIA states "up to 1.6 terabits per second (Tb/s) throughput" for ConnectX-9, "up to 800 gigabits per second (Gb/s) of data throughput" for ConnectX-8 and "400Gb/s throughput" for ConnectX-7. Port counts and PCIe generations aren't published there, so per-port rates can't be derived from this source.
2. AMD Pensando networking - "up to 800 Gbps of Ethernet connectivity" for Vulcano and "up to 400 Gbps Ethernet speeds" for Pollara 400.
3. Pollara 400 has a partner platform catalog, which implies it ships, but AMD doesn't state availability on that page.
4. The InfiniBand generation pairing isn't on NVIDIA's adapter page - it's taken from the platform rows in the node table above, where DGX B300 XDR uses eight ConnectX-8 at 800Gbps and DGX H100 NDR uses eight ConnectX-7 at 400Gbps.
5. Pre-release as of 2026-07-30. AMD's Vulcano numbers come from "AMD Engineering silicon modeling and AMD synthetic benchmark simulation" and "may vary when actual product(s) are released in market".
6. AMD advertises "up to 2.4 Tbps of scale-out bandwidth per GPU" for Vulcano. That's three 800Gbps NICs attached to one GPU - a platform configuration, not a single faster NIC. Same trap as the node-aggregate columns elsewhere in this chapter.
7. These two sit under `Announced, availability not confirmed` because the vendor page names the product but states no availability, not because the vendor says it is unreleased. NVIDIA lists ConnectX-9 in its adapter portfolio without a ship date; AMD gives Pollara 400 a partner platform catalog, which implies it ships, but says so nowhere. Neither is claimed here as available or unavailable.
8. Cornelis CN6000 SuperNIC adapters - Bandwidth `800G`; Performance `1.6Tbps bidirectional (800Gbps Tx and 800Gbps Rx)`; Dual QSFP-DD (`x4` @ 112 Gbps/lane); PCIe 6.0 x16. The throughput column is unidirectional, same convention as CN5000 / ConnectX.
9. Pre-GA; see inter-node notes 13-14 (product sampling in 2026, GA target Q4-2026).

InfiniBand

InfiniBand (IB) has been around for a few decades so there are many available configurations that can be found out there. So that if someone says they have InfiniBand that is insufficient information. What you need to know is the signaling rate and the number of IB links.

InfiniBand is a complete network protocol that implements RDMA (bypasses TCP/IP).

The cards themselves are in Network adapters above - ConnectX-7 for NDR, ConnectX-8 for XDR.

HDR and EDR are marked GA below because plenty of clusters still run them, but NVIDIA no longer lists a switch platform for either - its InfiniBand page names only Quantum-2 and Quantum-X800. So treat HDR/EDR as hardware you may inherit rather than hardware you would specify. Worth knowing too that InfiniBand has consolidated to very few manufacturers, which is a lock-in consideration when weighing it against RoCE - see the note on manufacturers further down in this section. As of 2026-07-28, XDR is the latest standardized InfiniBand generation. InfiniBand generation names describe a lane rate; a full-width port combines four lanes. They do not specify how many adapters or ports a node has.

Sorted by 4x port rate descending:

Generally available:

Generation	Rate/ lane (Gbps)	Lanes/ port	4x port rate (Gbps)	Ref.
XDR	200	4	800	1
NDR	100	4	400	2
HDR	50	4	200	3
EDR	25	4	100	3

Announced, availability not confirmed:

Generation	Rate/ lane (Gbps)	Lanes/ port	4x port rate (Gbps)	Ref.
LDR	800	4	3200	3
GDR	400	4	1600	3

Sources:

1. InfiniBand Trade Association XDR specification release
2. NVIDIA NDR cabling guide
3. InfiniBand Trade Association roadmap

Only 4x ports are shown in the product-oriented table because four lanes are the full-width port configuration used by the modern high-end products compared here. The 1x value is already the Rate/lane column. The earlier 8x and 12x arithmetic is preserved separately below so lane totals aren't compared with port or node totals as if they were the same scope.

GDR and LDR are roadmap targets rather than standardized generations; the table does not imply an availability date.

The June 2021 IBTA roadmap targeted GDR for 2025 and LDR some years later. This historical schedule is retained for context; it was a roadmap target, not confirmation of standardization or product availability.

The following table restores the earlier theoretical width arithmetic. It is sorted by **Links** ascending. These values are lane-rate multiplication, not a statement that a corresponding product or node configuration is available.

Links	EDR	HDR	NDR	XDR	GDR	LDR	Ref.
1	25	50	100	200	400	800	1
4	100	200	400	800	1600	3200	1
8	200	400	800	1600	3200	6400	1, 2
12	300	600	1200	2400	4800	9600	1, 2

Sources:

1. InfiniBand Trade Association roadmap
2. InfiniBand Trade Association historical 8X and 12X roadmap

The earlier LDR 8x value was 4800Gbps; the corrected lane-rate arithmetic is `8 * 800 = 6400Gbps`.

Latency in usecs, with columns ordered by generation from oldest to newest:

EDR	HDR	NDR	XDR	GDR	LDR
0.5	0.6	??	??	??	??

`??` = NDR and later didn't publish latency data

InfiniBand provides RDMA.

Here are some examples of NVIDIA devices with the fastest IB:

- One configuration of NVIDIA DGX H100 comes with 8x NVIDIA ConnectX-7 (CX7) Ethernet/InfiniBand ports each of 400Gbps, for a total of 3.2Tbps (0.4TBps) unidirectional to connect with other DGX servers.
- For DGX H100 SuperPOD the ConnectX-7s across all 32 DGX servers and associated InfiniBand switches provide 12.8TBps of unidirectional bandwidth (25.6TBps counting both directions) for use within the pod or for scaling out to multiple SuperPODs - that is an equivalent of 0.4TBps (3.2Tbps) unidirectional per node, matching the per-node figure above.
- NVIDIA DGX GB200 NVL72 compute trays provide four single-port ConnectX-7 interfaces at up to 400Gbps each, corresponding to one 400Gbps connection per GPU (networking documentation). Quantum-2 carries NDR at 400Gbps per port; an 800G OSFP cable can split into two 400Gbps ports, so it isn't a single 800Gbps NDR endpoint.
- NVIDIA DGX B300 systems provide eight ConnectX-8 interfaces at up to 800Gbps each. In the documented SuperPOD design, the compute fabric uses Quantum-X800 XDR, while the storage fabric uses Quantum-2 NDR.

InfiniBand used to have multiple manufacturers, but at the moment it's just NVIDIA (purchased Mellanox) - which makes it a single-vendor interconnect, a lock-in consideration when weighing it against Ethernet. Intel is often still counted as the second vendor because it purchased QLogic's InfiniBand business, but that lineage became Omni-Path - a separate fabric rather than InfiniBand - and Intel later sold it on to Cornelis Networks. Also see InfiniBand Trade Association.

Practical links:

- InfiniBand Utilities (the link could be outdated as it's versioned) - these are useful when debugging an IB setup.

Switch platforms

The switches the adapters plug into. Both fabrics are here so they can be compared directly - the InfiniBand pair first, then

Ethernet.

NVIDIA Quantum-2 InfiniBand

NVIDIA Quantum-2 InfiniBand Platform is the switch side of NDR, pairing with the ConnectX-7 adapters in Network adapters. It supports 400Gbps per port, provides RDMA, includes in-network computing with SHARP, and supports PCIe-5.

The switches provide 64 ports at 400Gbps or 128 ports at 200Gbps.

NVIDIA Quantum-X800 InfiniBand

NVIDIA Quantum-X800 InfiniBand Platform is the switch side of XDR, pairing with ConnectX-8, and supports 800Gbps per port and includes in-network computing with SHARP v4.

As of 2026-07-28, Quantum-X800 is used by Blackwell Ultra systems such as DGX B300 and GB300 NVL72; it is not specific to Rubin.

Spectrum-X Ethernet

Spectrum-X is NVIDIA's Ethernet answer to its own InfiniBand line, for shops that want RoCE rather than IB. NVIDIA describes it as "the tight coupling of the NVIDIA Spectrum-X Ethernet switch and the NVIDIA Spectrum-X Ethernet SuperNIC" and claims "1.6x over off-the-shelf (OTS) Ethernet".

Switch throughput is fabric capacity - it is not per-node injection bandwidth, and the two should never be added together or compared. Sorted by total throughput descending, then by height ascending:

Generally available:

Switch	Generation	Ports	Total throughput (Tbps)	Height
SN5600	Spectrum-4	64x OSFP 800GbE	51.2	2U
SN5600D	Spectrum-4	64x OSFP 800GbE	51.2	2U
SN5610	Spectrum-4	64x OSFP 800GbE	51.2	2U
SN5400	Spectrum-4	64x QSFP-DD 400GbE	25.6	2U

Announced, availability not confirmed:

Switch	Generation	Ports	Total throughput (Tbps)	Height
SN6800-LD	Spectrum-6	512x MMC-12 co-packaged optics	409.6	5U
SN6200-LD	Spectrum-6	32x OSFP 2x800GbE + 256x 200G backplane	102.4	1U
SN6600-LD	Spectrum-6	64x OSFP 2x800Gbps (128 ports)	102.4	2U
SN6810-LD	Spectrum-6	128x MMC-12 co-packaged optics	102.4	2U
SN6600	Spectrum-6	64x OSFP 2x800Gbps (128 ports)	102.4	3U

Source: NVIDIA Ethernet switching, as of 2026-07-30. NVIDIA does not state per-model availability there, so `GA` is `?` for the Spectrum-6 SN6000 systems - the newer co-packaged-optics generation - and `Y` for the Spectrum-4 SN5000 line, which has been shipping since the Spectrum-X launch.

Two more pieces worth knowing:

- Spectrum-XGS is the cross-datacenter variant, "built for scaling across multiple disparate data centers - located in different buildings or separated by hundreds of kilometers", claiming "1.9x higher NCCL performance in cross-data center environments".
- Multiplane scaling splits "each GPU's SuperNIC across two or more independent network planes" to reach "up to 128K GPUs in two tiers - 64x more than single-plane networks". So a single GPU's inter-node bandwidth may be spread over several fabrics rather than one.

Co-packaged silicon photonics

The direction both fabrics are heading: put the optical engine inside the switch package instead of using pluggable transceivers. NVIDIA frames it as "New Co-Packaged Silicon Photonic Networking Switches to Scale to Millions of GPUs, Multi-Site AI Factories", and for the Ethernet side claims "5x better network power efficiency" and "5x longer sustained AI application runtime" against pluggable optics. It is already in shipping part numbers - the Spectrum-6 `SN6800-LD` and `SN6810-LD` above use `MMC-12` co-packaged optics rather than OSFP cages.

Note that NVIDIA does not attach a Quantum product name to the InfiniBand version on its InfiniBand page - only the generic photonics framing - so there is no generation to put in a table yet. Power is the reason this matters: at 800Gbps and beyond, the transceivers become a serious fraction of a switch's power budget, and that is a datacenter-level constraint rather than a benchmark one.

Reaching beyond the rack

Not everything is a NIC or a top-of-rack switch. Three categories worth knowing exist, none of which you would size with the node and adapter tables:

- MetroX - InfiniBand long-haul, reaching "up to 40 kilometers". This is the InfiniBand answer to Spectrum-XGS for splitting one training job across buildings or sites
- InfiniBand routers - for joining separate InfiniBand subnets, which is how very large fabrics are partitioned
- InfiniBand-to-Ethernet gateways - for reaching Ethernet-attached storage and services from an InfiniBand compute fabric

If a job has to span sites, this is the layer that decides whether it is possible at all, and the latency it adds dwarfs anything discussed in the intra-node sections.

EFA

Elastic Fabric Adapter (EFA) is a recent inter-node networking technology created by AWS.

- EFA v1 0.4Tbps (effective 340Gbps for all_reduce tests) (P4 AWS instances)
- EFA v2 3.2Tbps (since Q3-2023, P5 AWS instances - 32 100GbE (4x28G) NICs!)
- EFA v3 3.2Tbps (since Q1-2025, P5en AWS instances - 16 200GbE (4x56G) NICs! and Trn2 AWS instances) - same theoretical speed as v2, but should be delivering a much better actual speed at real world message sizes.
- EFA v4 3.2Tbps (P6-B200 AWS instances - 8x 400Gbps NICs, i.e. 400Gbps per accelerator)
- EFA v4 6.4Tbps (P6-B300 AWS instances, since Q4-2025 - 16x 400Gbps NICs, i.e. 800Gbps per accelerator). The 2x over B200 comes from doubling the NIC count at the same per-NIC rate, which AWS attributes to PCIe Gen6 - so don't assume "EFA v4" implies 6.4Tbps.

The generation labels above are AWS's own: its supported instance types page groups every EFA-capable instance under a `Nitro v3 (EFA v1)` through `Nitro v6 (EFA v4)` heading. The per-instance bandwidth figures come from the separate

network specifications page, so the two pages are the pair you need.

footnote: the EFA generation tracks the Nitro generation, not the accelerator - and it says nothing about per-accelerator bandwidth. `p6e-gb200.36xlarge` sits under `Nitro v5 (EFA v3)` while `p6-b200.48xlarge` and `p6-b300.48xlarge` are under `Nitro v6 (EFA v4)`, yet the GB200 instance is the faster of the two per accelerator: AWS lists 3200Gbps for its 4 B200s, i.e. 800Gbps (100GBps) each, against the same 3200Gbps spread across 8 on P6-B200, i.e. 400Gbps (50GBps) each. So a higher `v` implies neither newer silicon above it nor more bandwidth per accelerator.

To count the EFA devices on a node:

```
fi_info -p efa -t FI_EP_RDM | grep -c provider
```

Divide the result by two. `libfabric` reports two providers per EFA device - one with fabric `efa-direct`, matching the NIC's raw capabilities (8KiB max send, no ordering, no matching), and one with fabric `efa`, offering the fuller interface that MPI or NIXL would use. So `16` means 8 NICs.

Do not count `/sys/class/infiniband/*` instead, because it also lists non-EFA adapters - a P6-B200 node shows 10 entries: the 8 EFA devices (`rdmap*`, each reporting `rate=400 Gb/sec`) plus 2 unrelated 100Gbps adapters. It is however the place to read the per-NIC rate: `cat /sys/class/infiniband/rdmap*/ports/1/rate`. `nvidia-smi topo -m` is the wrong tool for this - it is an intra-node tool and knows nothing about the inter-node fabric.

Gaudi2 (inter-node)

According to Gaudi2 spec, these nodes provide `3*8=24` NICs of 100GbE RoCE v2 RDMA for a total of 2.4Tbps of inter-node connectivity with other Gaudi2 nodes.

Gaudi3 (inter-node)

According to Gaudi3 spec, these nodes provide `3*8=24` NICs of 200GbE RoCE v2 RDMA for a total of 4.8Tbps of inter-node connectivity with other Gaudi3 nodes.

HPE Slingshot interconnect

HPE Slingshot interconnect seems to be used by HPCs. As of 2026-08 it provides 200Gbps per link. Some HPCs use 4 of those links to build 800Gbps interconnects, and, of course, with more links will deliver a higher overall bandwidth.

GPUDirect-TCPX

GPUDirect-TCPX is an RDMA-over-TCP software solution developed by Google for A3 instances. GPUDirect-TCPXO is a hardware-accelerated extension of TCPX available only on A3 Mega. The docs are scarce, but here is some information on TCPX and TCPXO.

This technology didn't catch on and has been phasing out while getting replaced with RoCE starting with Blackwell instances at GCP.

Omni-Path

Omni-Path Architecture (OPA). Originally by Intel, the technology got sold to Cornelis Networks. It's also known as Omni-Path Express (OPX).

case study: I used this technology at JeanZay HPC in France in 2022. It was only 135Gbps and while the vendor tried to fix it a year later it was still the same speed. Hopefully the issue has been resolved and the speed is much faster nowadays. Because it was so slow we had to use Megatron-DeepSpeed for training BLOOM-176B instead of the much easier to use DeepSpeed ZeRO).

Cornelis Omni-Path Accelerated Host Fabric Adapter CN-100HFA 100Gbps NICs have been around for many years now - and until 2025 this was the only Omni-Path generation that shipped, since Intel canceled the planned 200Gbps `OPA 200` series in July 2019. At 100Gbps per NIC you were unlikely to see Omni-Path offered for ML workloads unless someone installed many NICs per node.

CN5000 400Gbps NICs began shipping in June 2025 and have been broadly available since Q3-2025 - see note 18 under the inter-node networking table. One MI300X setup uses 8x of these for 3200Gbps of total unidirectional inter-node bandwidth.

footnote: Cornelis's marketing pages sometimes say "400Gbps bidirectional", but the product-family docs state "400 Gbps (4 x 100 Gbps) bandwidth **in each direction**" - so this chapter treats 400Gbps as unidirectional, the same way it treats ConnectX-7 NDR / EFA 400G.

CN6000 is the next generation - an 800Gbps SuperNIC on PCIe 6.0 that runs Omni-Path, RoCEv2, and Ultra Ethernet concurrently (each of its dual QSFP-DD ports independently configurable), so one card can speak Omni-Path on one port and RoCE/UEC on the other. Cornelis targets GA in Q4-2026; until then it sits in the announced half of the node and adapter tables.

footnote: Cornelis also writes "800 Gbps of bidirectional bandwidth" in marketing copy; the adapter page's own Performance line is `1.6Tbps bidirectional (800Gbps Tx and 800Gbps Rx)`, so this chapter treats 800Gbps as unidirectional - same call as the CN5000 footnote above.

Omni-Path provides RDMA.

Ultra Accelerator Link (UALink)

The UALink initiative is an attempt to create an open standard to compete with NVLink. Supposedly it'll be based on AMD's Infinity Fabric. The consortium published the `UALink 200G 1.0` specification in April 2025 and four more in April 2026, covering in-network compute, chiplets, manageability and 200G performance. As of 2026-08 I'm not aware of any actual hardware to speak of - the consortium lists specifications and members, but no products.

UB Link (UnifiedBus)

Huawei's scale-up interconnect, playing the same role for Ascend accelerators that NVLink does for NVIDIA and Infinity Fabric does for AMD. Worth knowing exists even if you can't buy it, because it is the third serious scale-up fabric and the one the UALink effort is implicitly racing.

Huawei's English pages quote it only at rack level, but the Chinese ones publish the per-accelerator figure. For the Atlas 950 SuperPoD they list a per-cabinet total interconnect bandwidth of up to 64 x 1.68TBps bidirectional. So each NPU gets **1.68TBps duplex, i.e. 840GBps unidirectional**, which puts UB Link between NVLink 5 (900GBps) and NVLink 4 (450GBps).

Three published figures corroborate each other, which is why the number is trustworthy despite the sourcing difficulty: the SuperPoD holds up to 1024 Ascend 950DT accelerators across 16 compute cabinets, which is 64 per cabinet exactly as the bandwidth line states; and 1024 x 1.68TBps is 1.72PBps, exactly the "total bandwidth up to 1.72PBps" the same page claims.

Architecturally it is a switched fabric, not point-to-point: the rack is listed as 16 compute cabinets plus 4 LingQu interconnect cabinets, all 44OU - so **four cabinets are dedicated purely to interconnect**. LingQu is Huawei's name for the UnifiedBus fabric, and those four cabinets are what make a 1024-accelerator scale-up domain possible. Huawei also claims link-level resilience - LingQu link flash-break self-recovery with transport-layer retransmission - and 2+2 optical path protection across cabinets.

caveat on sourcing: use Huawei's Chinese pages, not the English ones. Both are JavaScript applications, but the Chinese product pages inline the full specification table while the English equivalents inline almost nothing - the English Atlas 950 page never states a per-NPU UB Link figure at all. Everything quoted above is from the Chinese Ascend site, translated here. Treat any second-hand UB Link number with suspicion until you can trace it to Huawei.

Other essential network technologies

SHARP

NVIDIA Scalable Hierarchical Aggregation and Reduction Protocol (SHARP) - allows performing data reductions and aggregations on the network itself (in-network computing). This is very useful if you do a lot of MPI, NCCL and other network collectives that support SHARP, as those should get their latencies much improved.

To understand the importance of this technology - for `all-reduce` operations, instead of 2N sends, it will only need N+1 sends - so for a large N - it almost doubles the effective all-reduce throughput. (N is the number of communicating ranks/gpus). For details see all-reduce operation compatibility (you'd have to scroll down to get to that section).

Recent NCCL versions will automatically use this technology if it is available via `NCCL_ALGO=NVLS`. Practically at the moment the intra-node `all-reduce` bandwidth is improved by 30%, and inter-node `all-reduce` by about 25%. Both of those figures are NVLink SHARP - the slide further down puts the 30% at NVLink 4 inside a node and the 25% across an NVL72 domain, so even the inter-node case there stays within a scale-up fabric. SHARP in the InfiniBand switches is a separate path, the one that accelerates collectives crossing the scale-out network - the two share a name but not a mechanism.

The SHARP hardware, that is part of the NVSwitch or InfiniBand switches and also NVLink 4 and higher, includes arithmetic logic units (ALU) that perform the compute directly rather than using GPUs. It's said that it can perform math in FP64, FP32, FP16 and BF16 dtypes.

case study: I discovered SHARP accidentally when an H100 intra-node NVLink 4.0 all-reduce benchmark reported 480GBps for a 4GiB payload when the theoretical spec was only 450GBps! We figured out it's because NCCL turned on the new `NVLS` algo, which engaged NVLink SHARP. I still don't understand how it clocked speed faster than what the physical medium allows. I'm pretty sure that `busbw` calculation algorithm needs to be adjusted there from 2N to N+1 to get the real speed. There is a detailed discussion about this here. Bottom line: `busbw` may or may not be giving you the real bandwidth number depending on the `algo` NCCL chose to use, where only when `Ring` algo is used the `busbw` is correct.

To take advantage of this great feature:

- the collective has to engage enough GPUs that NCCL actually selects `NVLS` - see the measurements below; the switch is above 4 on H200 and above 5 on B200, so measure yours rather than assuming either number.
- ensure that the env var `NCCL_NVLS_ENABLE` is either unset or set to `1`.

Measured on an 8x H200 node at an 8GiB payload with `nccl-tests`, running each GPU count twice - once as-is and once with `NCCL_NVLS_ENABLE=0` to force the ring - so the last column isolates what SHARP is actually contributing:

GPUs	algo NCCL picks	busbw	busbw, NVLS off	gain from SHARP
4	Ring	363.3GBps	363.3GBps	1.00x
5	NVLS	414.2GBps	364.4GBps	1.14x
6	NVLS	435.6GBps	365.1GBps	1.19x
7	NVLS	460.0GBps	365.7GBps	1.26x
8	NVLS	473.1GBps	365.6GBps	1.29x

So it isn't a cliff at 8 GPUs, it's a ramp that starts above 4. At 4 GPUs NCCL doesn't even select `NVLS`, and the ring number is identical either way - exactly 1.00x. From 5 GPUs up it selects `NVLS` and the gain grows with each accelerator added. Admittedly a 5- or 7-GPU collective is an odd thing to run, but it does mean a partial-node job gets a partial benefit rather than none.

The same sweep on an 8x B200 `p6-b200.48xlarge` node (`nccl=2.27.7`, 8GiB, measured 2026-08-09) shows the ramp is not portable across generations - NCCL stays on `Ring` at 5 GPUs and only switches to `NVLS` from 6 up, confirmed with `NCCL_DEBUG=INFO NCCL_DEBUG_SUBSYS=INIT,TUNING`:

GPUs	algo NCCL picks	busbw	busbw, NVLS off	gain from SHARP
4	Ring	673.2GBps	673.8GBps	1.00x
5	Ring	677.8GBps	677.8GBps	1.00x
6	NVLS	778.2GBps	677.8GBps	1.15x
7	NVLS	811.2GBps	680.5GBps	1.19x
8	NVLS	838.0GBps	682.2GBps	1.23x

At full node the gain is still real - 1.23x here against H200's 1.29x - but a 5-GPU job on this B200 node gets none of it. The chooser on this box is AWS's `NCCL_TUNER_PLUGIN=ofi`, whose log line reads `base Tuner is chosen for platform: p6-b200.48xlarge`, so do not read the H200 threshold across to a different instance type.

NVIDIA's own documentation draws a 4-GPU line, though it takes one step to see why: NVLink SHARP is implemented with multicast - the switch reduces and then fans the result back out to every participant - which is why NCCL's setup log for `NVLS` literally reads `Created Multicast group`. So a statement about multicast is a statement about SHARP, and the GB200 NVL Partition User Guide says that "partitions that have less than or equal to four GPUs will not benefit from multicast and can accomplish all traffic through unicast" - matching the H200 boundary above, and sitting one GPU below the B200 one.

footnote: the multicast group itself is created regardless - `NCCL_DEBUG=INFO NCCL_DEBUG_SUBSYS=NVLS` reports `NVLS Created Multicast group` at 4 GPUs just as it does at 8, along with the same 16 nvls channels. What changes above the threshold is that NCCL starts *using* the `NVLS` algorithm. So "is a multicast group set up" and "is SHARP doing anything for you" are separate questions.

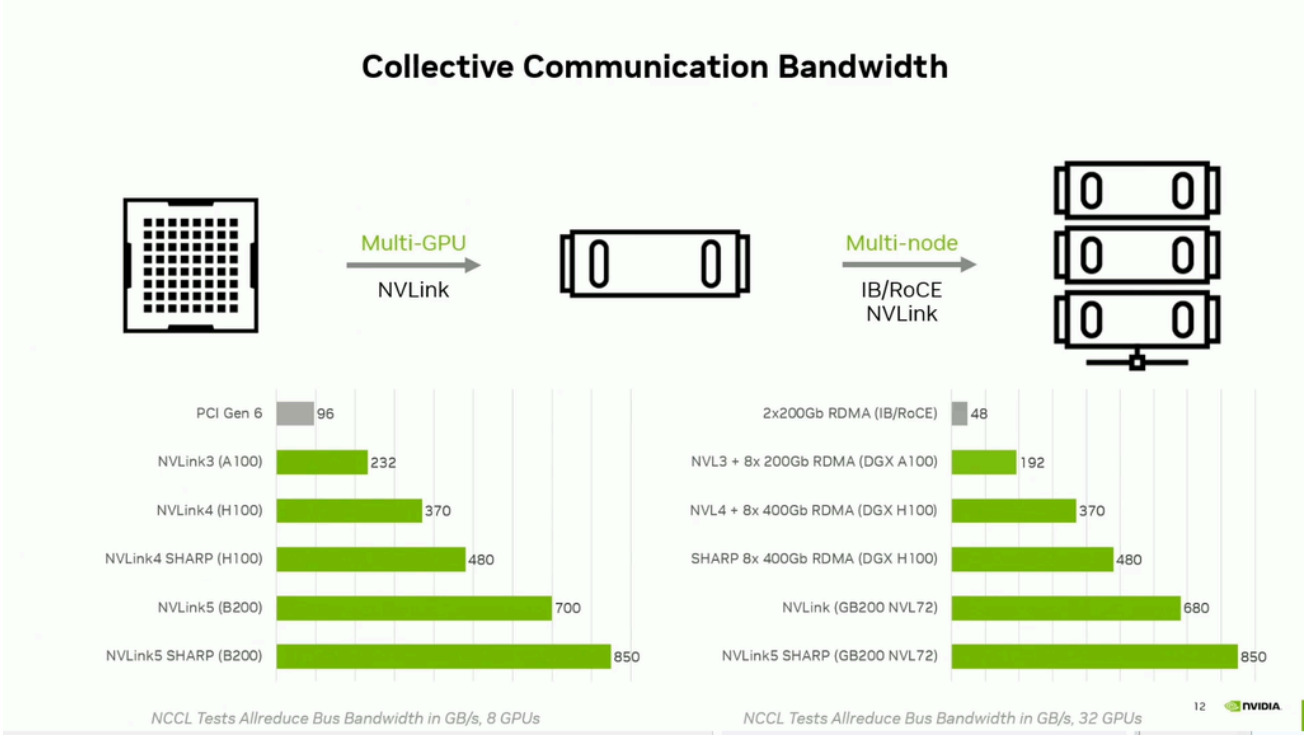
footnote: don't confuse the GPU count with `multicastGroupsLimit`, which the same guide says "must be 0 or a multiple of 4". That one is the number of multicast *teams* reserved to a partition out of the 1,024 the system provides - a fabric-manager resource allocation, unrelated to how many GPUs your collective engages.

In the case of NVL36, NVL72 and others bigger than NVL8 the granularity is likely 4 GPUs rather than 8, since a compute tray holds 2 GB200 modules and each module is 1 Grace CPU + 2 Blackwell GPUs. The NVIDIA GB200 NVL Partition User Guide points the same way: its partition sizing examples step in fours (72, 68 and 64 GPUs), and it states that "partitions that have less than or equal to four GPUs will not benefit from multicast and can accomplish all traffic through unicast". Multi-cast is a requirement for NVLink SHARP to work; for more clarity on why multi-cast is needed, see this.

This one still needs validating on real NVL hardware. The two HGX sweeps above already disagree with each other (H200 switches at 5, B200 at 6), and the guide's statements concern fabric-level partitioning rather than which algorithm NCCL selects inside a partition - so the NVL case could land on either, or on neither. Read the 4-GPU granularity as the likely case from the docs rather than an established one, until someone runs the sweep on an NVL36 or NVL72 system.

Part of what makes this murky is that the GB200 use case is ambiguous/confusing with regards to counting GPUs, since 1x GB200 == 2x B200 + 1x CPU, therefore the NVIDIA doc talks about 4x GB200, which is 8x B200 - and reading that "4x" as a minimum partition size is an easy way to arrive at a granularity of 8 GPUs when the tray itself holds only 4.

The left side of the following slide shows a nice 30% speed up of `all-reduce` bandwidth from NVLink 4 non-SHARP (370GBps) to NVLink 4 SHARP (480GBps). I was able to match the results with a payload of about 8GiB. For `all-reduce` on NVL72 (right side) it shows a 25% improvement (`850/680`).



source

To see it for yourself on your own node, run the same benchmark twice - `NCCL_NVLS_ENABLE=0` forces the ring path:

```
# SHARP (default)
torchrun --nproc_per_node=8 --rdzv_endpoint localhost:6000 --rdzv_backend c10d
all_reduce_bench.py
# ring
NCCL_NVLS_ENABLE=0 torchrun --nproc_per_node=8 --rdzv_endpoint localhost:6000 --rdzv_backend c10d
all_reduce_bench.py
```

On 8x H200 with `nccl=2.27.7` that gives 482.26GBps against 367.61GBps at a 16GiB payload - a 1.31x gain, matching the 480/370 on the slide above. Add `NCCL_DEBUG=INFO NCCL_DEBUG_SUBSYS=INIT,GRAPH,TUNING` to confirm which path was taken and to find where the switch happens: it reports `Algo RING proto LL` up to a 1MiB payload and `Algo NVLS proto SIMPLE` from 2MiB up, and `Algo RING proto SIMPLE` throughout when NVLS is disabled. `NCCL_DEBUG_FILE=/tmp/nccl.%h.%p.log` keeps that output out of the benchmark's own.

The GPU count is only one of the two conditions. The other is the collective itself - `NVLS` implements `all-reduce` and nothing else. Measured on the same node at a 16GiB payload, all with 8 GPUs engaged:

collective	algo	busbw	% of 450GBps
all-reduce, `NCCL_NVLS_ENABLE=1`	NVLS	480.0GBps	107%
all-reduce, `NCCL_NVLS_ENABLE=0`	Ring	367.2GBps	82%
all-gather	Ring	361.4GBps	80%
reduce-scatter	Ring	362.9GBps	81%

The second row is the same collective at the same width with SHARP switched off, and it is the one that makes the other two readable: `all-gather` and `reduce-scatter` are not slow collectives, they are simply running the ring path that `all-reduce` also falls back to. All three land at 80-82% of the unidirectional spec - the ordinary NVLink efficiency you would expect from Unidirectional vs Bidirectional (Duplex). Which means `~80%` is the normal case and the 107% is the exception, reachable only by an `all-reduce` wide enough that NCCL selects `NVLS`.

If you want to reproduce these numbers use nccl-tests:

```
NCCL_NVLS_ENABLE=1 ./build/all_reduce_perf     -b 16G -e 16G -g 8 -z 1  # 480.0 - NVLS
NCCL_NVLS_ENABLE=0 ./build/all_reduce_perf     -b 16G -e 16G -g 8 -z 1  # 367.2 - Ring, SHARP off
./build/all_gather_perf                        -b 16G -e 16G -g 8 -z 1  # 361.4 - Ring, wrong
collective
./build/reduce_scatter_perf                    -b 16G -e 16G -g 8 -z 1  # 362.9 - Ring, wrong
collective
```

footnote: `busbw` uses a different correction factor per collective - `2*(n-1)/n` for `all-reduce` against `(n-1)/n` for `all-gather` and `reduce-scatter` - and `nccl-tests` applies the right one for each, so the last column really is comparable across the rows as a fraction of the wire.

Understanding why inter-node network speed is of a huge importance

This is probably one of the most important multi-segment section that you really want to understand well. While it seeks out to show how important the inter-node speed is, to build up the case it'll teach on the way many important training-related concepts.

The basics

First, let's get a bit of a feeling what all those Gbps/GBps practically mean.

If your model is 80B parameter large, and you need to transmit every parameter or a gradient on the network even once in float32 (fp32) format, which requires 4 bytes per parameter, so you need to send 320GB (`80*4`) of data, or 2560Gb (`320*8`). If your network's bandwidth is 200Gbps it will take 12.8 seconds (`2560/200`) to transmit. And if you had 1600Gbps network then it'd take only 1.6 seconds.

1-GPU training

Let's start with a much smaller model of say 2B params, to train it you'd need at least 18 bytes per parameter in mixed half precision which requires 2 bytes to store. So 36GB (`18*2`) of memory just for model weights, optimizer states and gradients. Plus you need additional memory for activations. Saved hidden states, activation checkpoints, and logits scale linearly with batch size and sequence length. Standard dense attention compute scales quadratically with sequence length, while temporary attention memory also scales quadratically only when the kernel materializes the full attention matrix. Taking 80GB A100 GPU for example, we can definitely train this model on a single GPU.

We then assume for the moment that the DataLoader is fast enough to be negligible in duration compared to the compute time. And thus we get a close to a perfect MFU (Model FLOPs Utilization):

```
[DL][  compute  ][DL][  compute  ][DL][  compute  ]
-------------------------------------------------> time
|<--iteration-->||<--iteration-->||<--iteration-->|
```

which means that the GPU just needs to do many matmuls and it'd do it amazingly fast. In this situation you will get the highest ROI (Return on Investment).

Single node training

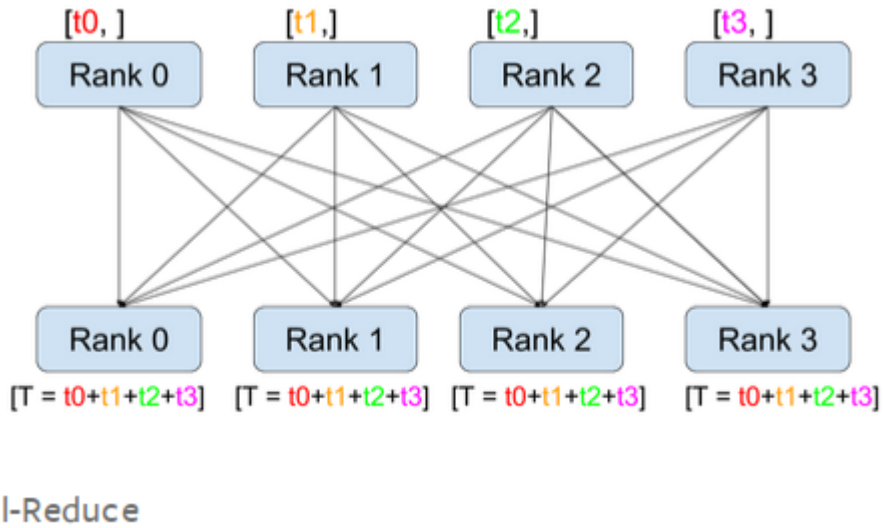
The previous situation was fantastic due to the close to perfect MFU, but you realize that the training on a single GPU is going to take quite some time, since we are in AI race you'd probably want to finish the training sooner than later. So you'd ask - can I train the model on 8 GPUs instead, and the answer would be - yes, of course. With one caveat - at the end of each iteration you'd need to sync the gradients between the 8 processes (each process for a single GPU), so that each participating process of the training can benefit from what the other 7 have learned during the last iteration.

footnote: You could, of course, use less than 8 GPUs, it is just that most NVIDIA GPU-based compute nodes these days have 8 GPUs so why not get the best return on investment. On NVLink 4 and higher there is a performance reason as well: an `all-reduce` over 4 or fewer accelerators loses SHARP altogether, and 5 to 7 accelerators get only part of its benefit.

footnote: In the ideal world the training on 1 GPU for 8 durations of time, should cost the same as training on 8 GPUs for 1 duration of time. That's what one would expect - the same $$ spent, but finishing 8x faster. But because of data synchronization requirements, this is not the case.

If the experimental model still contains 2B params like in the previous section and grads are in fp32 then the training program needs to send 8GB (`2B * 4B`) of data on every iteration. Moreover, since syncing the gradients requires an `all_reduce` collective - it needs to transmit the data twice - the first time sending the gradient data by each GPU, computing the sum of gradients and send this value back to each participating GPU so that each training process will benefit from the learning advancements each of its peers made in the last iteration.

Here is the all-reduce collective visualized:



All-Reduce

(source)

So we need to send 8GB twice on every iteration, which means we need to send 16GB of data.

footnote: and to be exact the 2x comms volume for all-reduce is really `2*(n-1)/n` where n is the number of participating GPUs. So if n=2, the coefficient is just 1 since `2*(2-1)/2=1` and 1.75 for n=8 since `2*(8-1)/8=1.75`. It becomes already very close to 2 at n=64.

footnote: there is also the important issue of latency of the network - which is multiplied several times due to how data is gathered from all participating GPUs. But, given that here we are moving a very large payload the latency contributes a

very small overhead and for simplicity can be ignored.

How long will it take to send 16GB of data?

- A100 @ 300GBps: `16/300` = 0.053 secs
- H100 @ 450GBps: `16/450` = 0.035 secs

which is incredibly fast!

And here is how our timeline will look like:

```
[DL][  compute ][comms][DL][  compute ][comms][DL][  compute ][comms]|
-------------------------------------------------------------------------> time
|<---- iteration ---->||<---- iteration ---->||<---- iteration ----->|
```

oh and this whole synchronization protocol is called DDP (DistributedDataParallel) in the PyTorch lingo.

Comms and compute overlap

Even with this really fast comms the network still creates a bottleneck and leads to a short idling of the GPUs. To solve this issue the advanced algorithms implement an overlap of comms and compute. Until now we approached the problem as one single transmission, but in reality each model is made of many layers and each layer can transmit the gradients it has computed, while the next layer is computing its gradients. So if you look at the level of the model, what happens in the `backward` path is:

```
[   compute   ][   compute   ][   compute   ]
               [comms]        [comms]        [comms]
-------------------------------------------> time
<- layer -1 ->|<- layer -2 ->|<- layer -3 ->|
```

so once the last layer (-1) computed its gradients it all-reduces them while the 2nd to last layer performs its `backward`, and so on, until the first layer finished with gradients and it finally sends its gradients out.

So now you understand how overlapping works. Now our timing diagram becomes very similar to the diagram we had for a single GPU:

```
[DL][  compute  ][DL][  compute  ][DL][  compute  ]
[  comms ]       [  comms]        [  comms]
-------------------------------------------------> time
|<--iteration-->||<--iteration-->||<--iteration-->|
```

and we hope that comms are faster than DL+compute, since if they aren't faster than we have the following GPU idling gaps:

```
[DL][  compute  ][idle][DL][  compute  ][idle][DL][  compute  ][idle]
[         comms       ][         comms       ][         comms       ]
-------------------------------------------------------------------------> time
|<---  iteration  --->||<---  iteration  --->||<---  iteration  --->|
```

When comms take longer than compute, the comms part that doesn't overlap is called "exposed communication". Here the compute is blocked waiting for the arrival of the data it needs to continue.

Calculating TFLOPS

Calculating TFLOPS answers the question of how long will it take to perform a compute.

There is a bit of nomenclature confusion here as TFLOPS as the final `s` sometimes means `sec` and at other times just `ops`.

For example, when you read, the A100 spec the TFLOPS there means TeraFloatingPointOperations per second.

So let's define these abbreviations exactly:

- TFLOPS - TeraFLoatingpointOPerations per Second (another way is TFLOP/s or TFLOPs/s)
- TFLOP - TeraFLoatingpointOPerations (or TFLOPs - lower case `s` but it's already confusing)

Also see the wiki page for more clarifications.

For GPT-family of decoder transformers models we can use the math described in this BLOOM-176 docs:

Here is how many TFLOP are processed per second:

```
tflops = model_size_in_B * 4 * 2 * seqlen * global_batch_size / (time_in_sec_per_iteration *
total_gpus * 1e3)
```

This formula assume one uses activation recomputation which saves GPU memory while introducing a smallish overhead. If one doesn't use it then replace `4` with `3` as the model has to do only 1x compute per `forward` and 2x per `backward` (since the grads are calculated twice - once for inputs and once for weights). With activation recomputation the `forward` is done twice and thus you have an additional path which leads to a multiplier of `4` instead of `3`

footnote: activation recomputation and gradient checkpointing both refer to the same technique.

so let's remove the time component, which will give us the total TFLOP

```
tflop = model_size_in_B * 4 * 2 * seqlen * global_batch_size / (total_gpus * 1e3)
```

So let's say we have:

- `seqlen=2048` (sequence length)
- `global_batch_size=16`

and we already defined:

- `total_gpus=8`
- `model_size_in_B=2`

This gives us:

```
tflops = 2 * 4 * 2 * 2048 * 16 / (8 * 1e3) = 65.536 TFLOP
```

So if we do a mixed half-precision training and most of the operations are done in half-precision then we can roughly say that we do 312TFLOPS on A100 and usually a well optimized framework on a well-tuned hardware will do at least 50% MFU - that is it'll be able to compute at about 1/2 peak performance.

footnote: It's a ~3x 989TFLOPS on H100 (scroll to the end) and also it shows a misleading 2x numbers for sparsity so you

have to mentally divide it by 2.

So continuing this train of thought it means that the setup will have about 156TFLOPS for mixed half-precision training - and so it'll take 0.42 secs on A100 GPU to process a single iteration (2x `forward` and 2x `backward` compute) if we ignore the overhead of the DataLoader (which we hope is close to instant).

Earlier we said that a typical A100 node has an intra-node NVLink connection of 300GBps, and thus we said that to send 16GB of grads will take `16/300` = 0.053 secs.

And we measured our compute to be 0.42 secs, so here the network isn't a bottleneck as `0.42 > 0.053` so the compute will be slower than communication.

You can now do several thought experiments - for example if you halve the batch size or the sequence length you will halve the compute time.

footnote: this is a very rough suggestions since GPUs work the fastest when the matrices they multiple are huge. But this is good enough for a simplified thought experiment we are having here. In reality halving the dimension will not halve the compute time.

OK, but hopefully at this point it's quite clear that if you remain at the boundaries of a single node, you don't need to worry about your GPUs idling.

But what if you want to speed up the training even more and throw say 4x 8-GPU nodes at it. (and of course you don't have a choice but to use multiple nodes if you have a much larger model). Suddenly, the comms can become an even bigger bottleneck.

Multiple node training

So here we are continuing with the idea of 2B param model and we will now use 32 GPUs across 4 nodes to speed up the training even more.

While each group of 8 GPUs is still connected with super-fast NVLink technology, the inter-node connections are usually in an order of magnitude slower.

Let's say you have a 200Gbps connection. Let's repeat the math from the previous section of how long it'll take to reduce 16GB of gradients.

16GB is 128Gb, and so at 200Gbps this will take 0.64 seconds.

And if stick to the compute taking 0.42 seconds, here we end up with comms taking longer than compute since `0.64 > 0.42`.

Let's bring both use cases together:

nodes	comms	compute	comms is a bottleneck
1	0.053	0.42	no
4	0.64	0.42	yes

on this 200Gbps inter-node setup the comms are 12x slower than the same performed on an intra-node NVLink connections.

footnote: this arithmetic assumes the whole payload crosses the inter-node wire at one accelerator's NIC rate - a flat ring across all 32 ranks. For example, on a 4-node 8x H200 cluster (`p5en.48xlarge`, NCCL 2.27.7, EFA) a 4GiB `all-reduce` across 32 ranks selects `Ring`, measured 2026-08-07. Forcing `NCCL_ALGO=allreduce:ring` reproduced the default to within 0.1% - 364.65 against 364.87GBps `busbw` - while `NCCL_ALGO=allreduce:nvlstree` was available but 15% slower at 310.07GBps, which is why the AWS tuner rejects it. Do not assume this generalizes: the algorithm is chosen per platform, and

on that cluster the chooser is AWS's `NCCL_TUNER_PLUGIN=ofi` plugin rather than NCCL's own tuner. Check yours with `NCCL_DEBUG=INFO NCCL_DEBUG_SUBSYS=INIT,TUNING`.

In this case even though we still have the much faster NVLink connection, we don't really benefit from it, since the whole ensemble communicates at the speed of the slowest link. And that slowest link is the inter-node connection.

So in this particular situation if you were able to get a 400Gbps inter-node the speed would double and the comms will finish in 0.32 secs and thus will be faster than that 0.42 secs the compute would take.

footnote: you will never be able to get the advertised speed fully on the application level, so if it's advertised as 400Gbps in the best case expect to get 320Gbps (about 80%). So make sure to take this into the account as well. Moreover, depending on the payload of each collective - the smaller the payload the smaller the actual network throughput will be.

And remember this was all handling a pretty tiny as considered these days 2B param model.

Now do the same math with 20B and 200B parameter model and you will see that you need to have a much much faster inter-node connectivity to efficiently scale.

Large model training

Of course, when we train large models we don't use DDP, because we simply can't fit the whole model on a single GPU so various other techniques are used. The details are discussed in a dedicated chapter on Model Parallelism, but the only important thing to understand immediately is that all scalability techniques incur a much larger comms overhead, because they all need to communicate a lot more than just gradients and therefore the amount of traffic on the network can easily grow 3x and more as compared to the DDP protocol overhead we have been exploring so far.

It can be difficult to do even approximate math as we did in this chapter, because the actual compute time depends on the efficiency of the chosen framework, how well it was tuned, how fast the DataLoader can feed the batches and many other things, therefore there is no standard MFU that one can use in the math and you will discover your MFU when you configure and run the first few steps of the large model training, and then you will read the Performance chapters and improve your MFU even more.

As I have shown in these sections it should be possible to be able to do a back-of-envelope calculations once you understand the specific scalability technique and its networking costs, so that you could know ahead of time which Inter-node network speed you need to require from your acquisition manager. Of course, you also need to understand the particular model architecture and calculate how many TFLOP it will take to do a single iteration.

Important nuances

Real network throughput

The network throughput in the advertised spec and the actual throughput will never be the same. In the best case you can expect about 80-90% of the advertised spec.

Then the network throughput will depend on the size of payload being sent during each communication. The higher the payload the higher the throughput will be.

Let's demonstrate this using nccl-tests on a single A100 node:

```
$ ./build/all_reduce_perf -b 32k -e 16G -f 2 -g 8 -n 50
[...]
           size    time   algbw   busbw
            (B)    (us)  (GB/s)  (GB/s)
         32_768   43.83    0.75    1.31
         65_536   46.80    1.40    2.45
```

```
        131_072   51.76    2.53    4.43
        262_144   61.38    4.27    7.47
        524_288   80.40    6.52   11.41
       1048_576   101.9   10.29   18.00
       2097_152   101.4   20.68   36.18
      4_194_304   101.5   41.33   72.33
      8_388_608   133.5   62.82  109.93
     16_777_216   276.6   60.66  106.16
     33_554_432   424.0   79.14  138.49
     67_108_864   684.6   98.02  171.54
    134_217_728  1327.6  101.10  176.92
    268_435_456  2420.6  110.90  194.07
    536_870_912  4218.4  127.27  222.72
  1_073_741_824  8203.9  130.88  229.04
  2_147_483_648   16240  132.23  231.41
  4_294_967_296   32136  133.65  233.88
  8_589_934_592   64074  134.06  234.61
 17_179_869_184  127997  134.22  234.89
```

footnote: I massaged the output to remove unwanted columns and made the size more human readable

This benchmark run an `all_reduce` collective for various payload sizes from 32KiB to 16GiB. The value that we care about is the `busbw` - this column tells us the real network throughput as explained here.

As you can see for payloads smaller than 8MiB the throughput is very low - and it starts saturating around payload size of 512MiB. It's mostly because of latency. Reducing a single 4GB payload is much faster than 1000x 4MB payloads.

Here is the same sweep on an 8x H200 node, using this repo's all_reduce_bench.py, which reports the same two columns:

```
$ python -u -m torch.distributed.run --nproc_per_node=8 all_reduce_bench.py

| payload |    busbw   |    algbw   |
| ------: | ---------: | ---------: |
|   32KiB |   1.44GBps |   0.82GBps |
|   64KiB |   2.93GBps |   1.67GBps |
|  128KiB |   5.73GBps |   3.27GBps |
|  256KiB |  11.74GBps |   6.71GBps |
|  512KiB |  23.94GBps |  13.68GBps |
|    1MiB |  39.93GBps |  22.82GBps |
|    2MiB |  64.53GBps |  36.87GBps |
|    4MiB | 107.16GBps |  61.23GBps |
|    8MiB | 155.53GBps |  88.87GBps |
|   16MiB | 219.89GBps | 125.65GBps |
|   32MiB | 275.84GBps | 157.62GBps |
|   64MiB | 346.51GBps | 198.01GBps |
|  128MiB | 401.93GBps | 229.67GBps |
|  256MiB | 436.15GBps | 249.23GBps |
|  512MiB | 450.29GBps | 257.31GBps |
|    1GiB | 463.58GBps | 264.90GBps |
|    2GiB | 469.17GBps | 268.10GBps |
|    4GiB | 473.10GBps | 270.34GBps |
```

```
|    8GiB | 477.45GBps | 272.83GBps |
|   16GiB | 482.26GBps | 275.58GBps |
```

The curve has the same shape - low until a few MiB, most of the way there by 512MiB - but the saturated end is instructive. Comparing the 16GiB rows against each node's unidirectional spec:

node	busbw @16GiB	uni-dir. spec	% of spec
A100, NVLink 3	234.89GBps	300	78%
H200, NVLink 4, ring	367.61GBps	450	82%
H200, NVLink 4, SHARP	482.26GBps	450	107%

The ~80% rule of thumb from Unidirectional vs Bidirectional (Duplex) is alive and well - the H200 node lands on 82% when NCCL rings, which is what the middle row measures with `NCCL_NVLS_ENABLE=0`. What takes the top row past the wire spec is SHARP: at large payloads NCCL reduces inside the NVSwitch, so fewer bytes cross the links than a ring would need and `busbw`'s ring-based formula stops describing the wire. The A100 node has no such path, which is why its number obeys the rule without any flag.

Here is a benchmark that demonstrates that: all_reduce_latency_comp.py. Let's run it on an 8x H200 node:

```
$ python -u -m torch.distributed.run --nproc_per_node=8 all_reduce_latency_comp.py

----------- 1x 4.0GB ----------------
 busbw: 3722.778 Gbps

----------- 1000x 0.004GB ----------------
 busbw: 1146.127 Gbps
```

It's easy to see that it's about 3x slower in this particular case to send the same payload but in 1000 smaller chunks.

footnote: this benchmark does 3 warmup iterations before timing anything, and it has to. NCCL sets up its channels and buffers on the first collective of a given size, and on NVLink 4 and higher also registers the NVLS multicast group - all of which would otherwise be charged to the `1x` measurement, since that one runs first. Without the warmup the same 4GB payload measured 561Gbps instead of 3722Gbps on this node, which is not only 6x low but large enough to reverse the comparison and make 1000 small chunks look faster than one big one.

So when you calculate how long does it take to `all_reduce` a given payload size, you need to use the corresponding `busbw` entry (after of course you have run this benchmark on your particular hardware/environment).

Figuring out the payload can be tricky since it'd depend on the implementation of the framework. Some implementations will reduce each weight's gradient alone which obvious would lead to a very small payload and the network will be very slow. Other implementations bucket multiple gradients together before reducing those, increasing the payload and minimizing the latency impact.

But let's go back to the benchmark results table. This test was done on an A100 node that runs NVLink advertised as uni-directional 300GBps so we get about 78% of the theoretical speed with 16GiB payload and more than that the benchmark crashes. It can be seen from the last few rows of the table that not much more can be squeezed.

We can also run p2pBandwidthLatencyTest which performs a low-level p2p benchmark.

Now let's run it on A100:

```
./p2pBandwidthLatencyTest
[...]
Unidirectional P2P=Enabled Bandwidth (P2P Writes) Matrix (GB/s)
   D\D     0      1      2      3      4      5      6      7
     0 1581.48 274.55 275.92 272.02 275.35 275.28 273.62 273.20
     1 274.70 1581.48 275.33 272.83 275.38 273.70 273.45 273.70
     2 274.81 276.90 1594.39 272.66 275.39 275.79 273.97 273.94
     3 273.25 274.87 272.12 1545.50 274.38 274.37 274.22 274.38
     4 274.24 275.15 273.44 271.57 1584.69 275.76 275.04 273.49
     5 274.37 275.77 273.53 270.84 274.59 1583.08 276.04 273.74
     6 275.61 274.86 275.47 273.19 272.58 275.69 1586.29 274.76
     7 275.26 275.46 275.49 273.61 275.50 273.28 272.24 1591.14
[...]
```

As you can see in the Unidirectional section of the report we do get 274GBps out of the advertised 300GBps (~91%).

Please note that when I re-run this same test on H100s (NVLink 4.0) I got a much worse efficiency:

```
Unidirectional P2P=Enabled Bandwidth (P2P Writes) Matrix (GB/s)
   D\D     0      1      2      3      4      5      6      7
     0 2494.51 364.13 375.99 378.03 376.77 376.71 374.85 375.66
     1 375.18 2533.95 376.08 374.98 376.21 375.96 375.76 375.12
     2 363.43 393.28 2532.67 376.35 377.14 376.47 375.76 375.48
     3 369.90 375.92 393.63 2525.38 376.58 375.88 376.13 377.01
     4 376.20 376.28 375.20 393.52 2526.02 375.82 375.05 376.10
     5 376.26 376.60 375.54 375.52 376.81 2521.18 376.37 376.60
     6 374.31 376.19 376.80 376.32 376.83 376.44 2529.85 376.39
     7 376.17 376.49 376.53 374.95 376.30 376.82 375.71 2519.78
```

So 376GBps out of 450GBps is 83% (not very good).

Bottom line - in this particular setup:

1. if you have huge payloads you will be able to use about 80% of the advertised 300GBps
2. if the payload of each communication is smallish it could be far far lower.

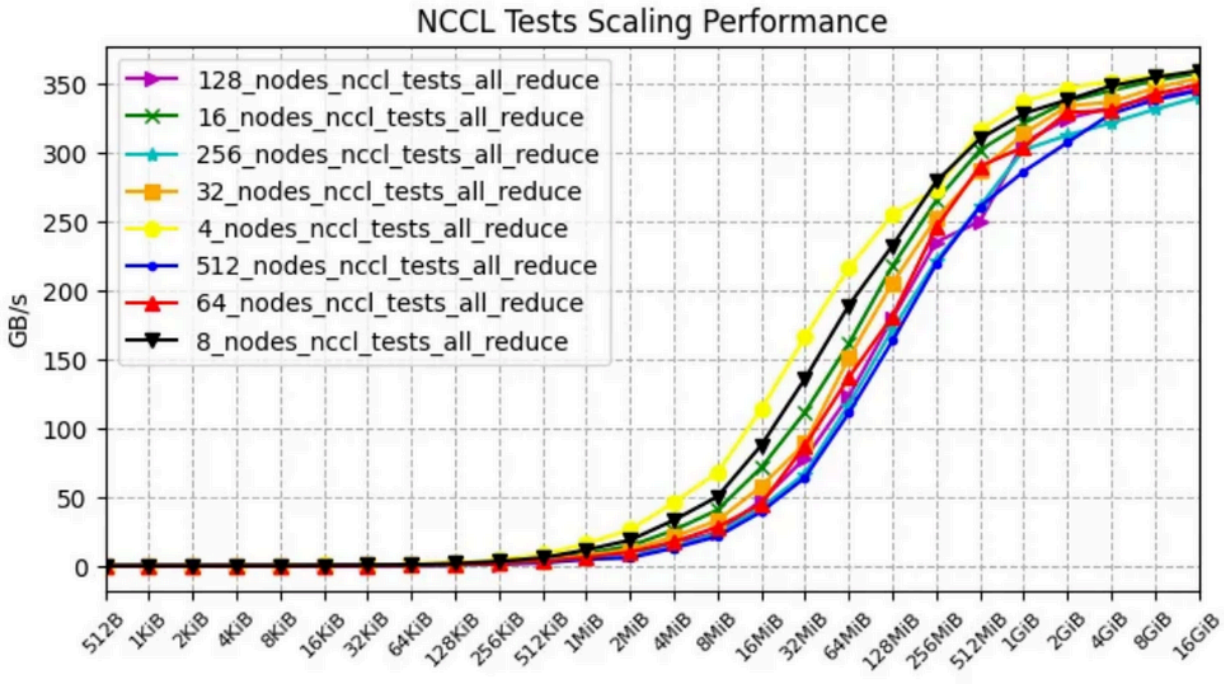
On GB200 (NVLink 5.0) (on a single nvl4 node w/ 4 GPUs):

```
Unidirectional P2P=Enabled Bandwidth (P2P Writes) Matrix (GB/s)
   D\D     0      1      2      3
     0 5714.94 746.74 746.56 748.08
     1 743.89 5820.72 747.18 746.83
     2 746.16 750.28 5814.63 746.71
     3 744.82 749.00 747.16 5816.66
```

746GBps out of 900GBps (82.8%) - very similar to NVLink4's efficiency.

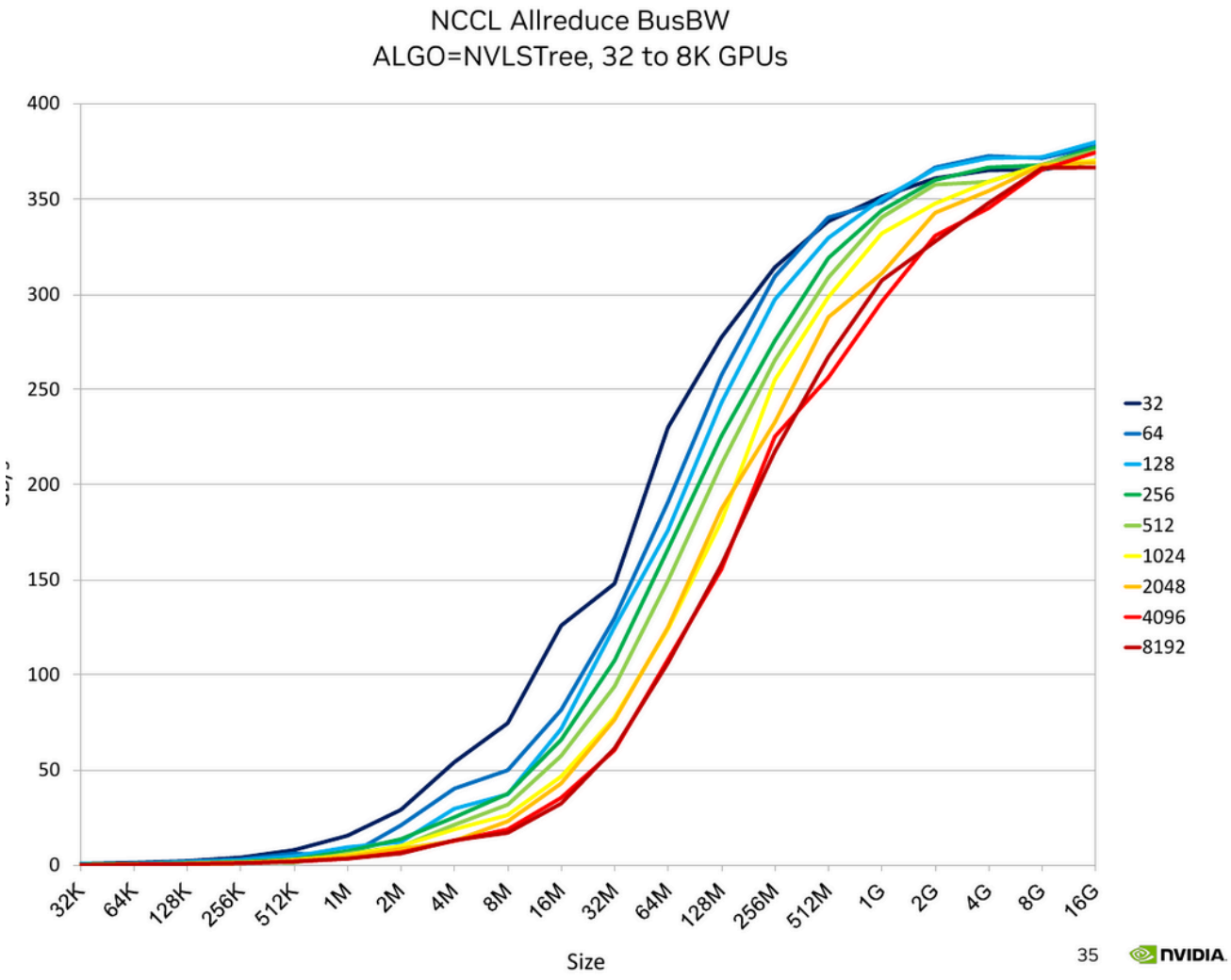
note: NVIDIA/nvbandwidth is supposed to be a more detailed and correct benchmark to replace `p2pBandwidthLatencyTest`, but I found the latter to provide very similar results.

The following plot demonstrates how the actual bandwidth changes for all-reduce with the size of the message and the number of participating nodes (4 to 512 nodes):



(source)

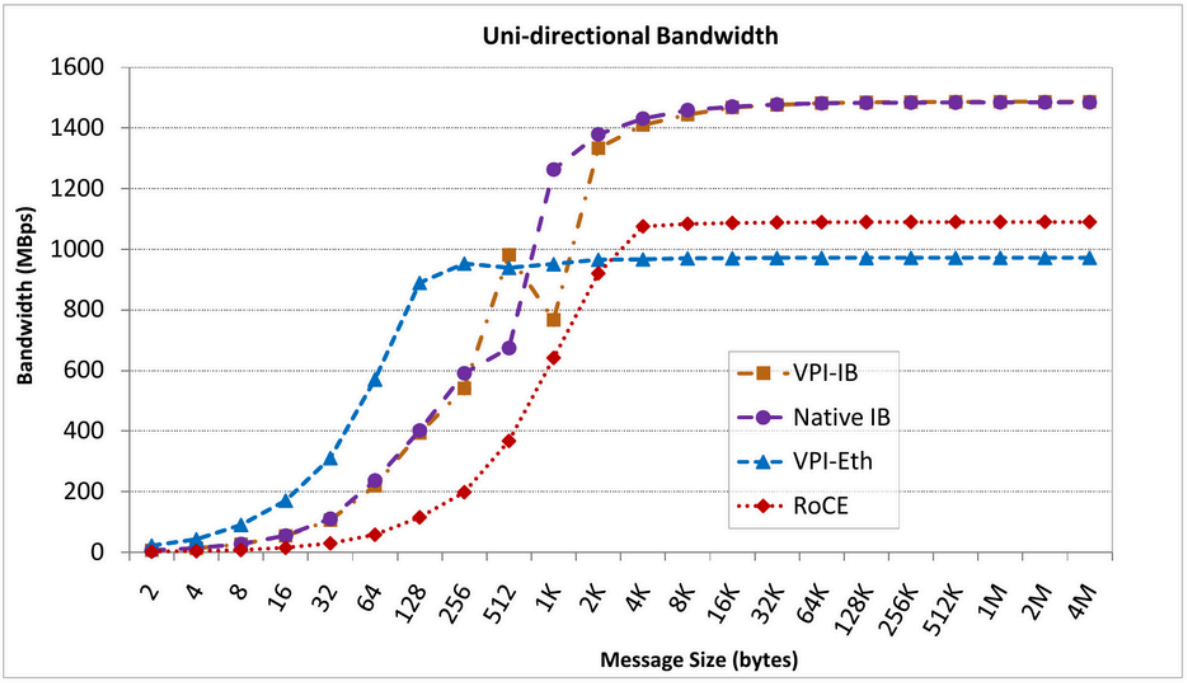
And here is a similar plot, but using NVLSTree algo, which helps to reach an even better performance on H100s (4 to 1024 nodes):



source

Here is another similar plot but it compares the message sizes and several networks:

Low-level Uni-directional Bandwidth Measurements

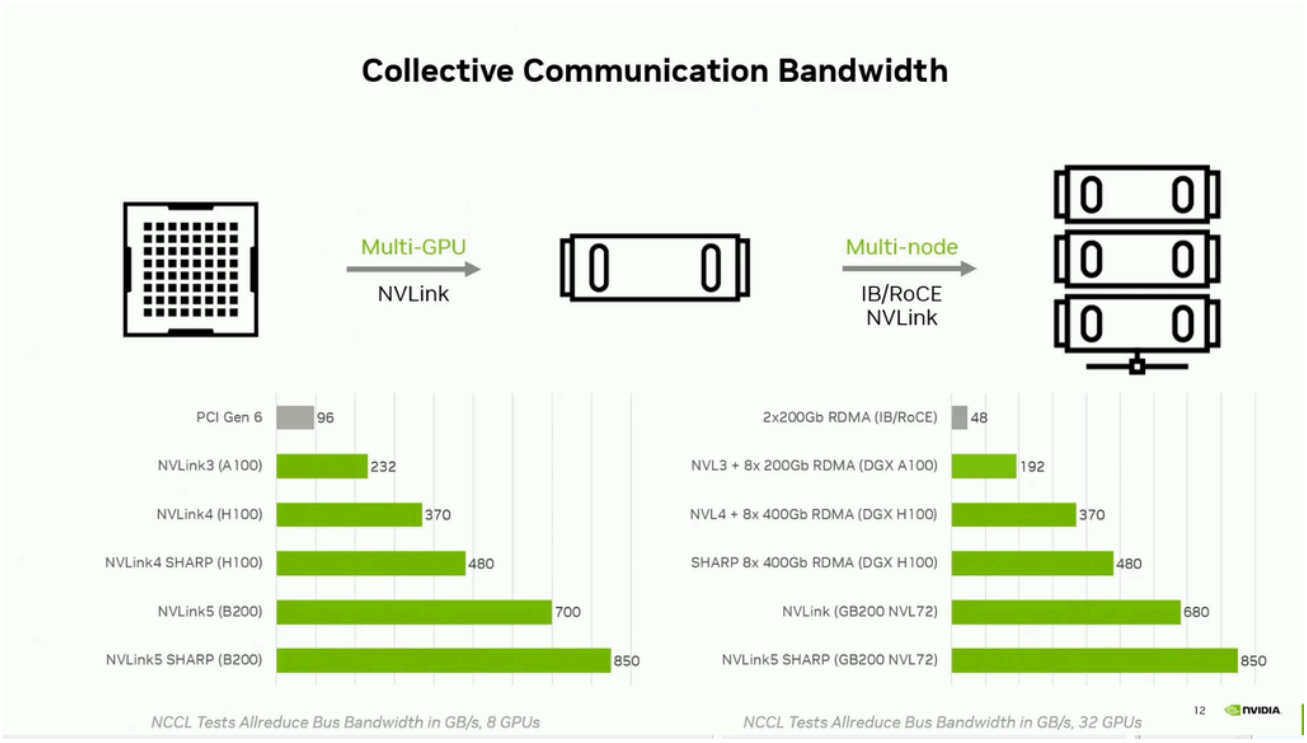


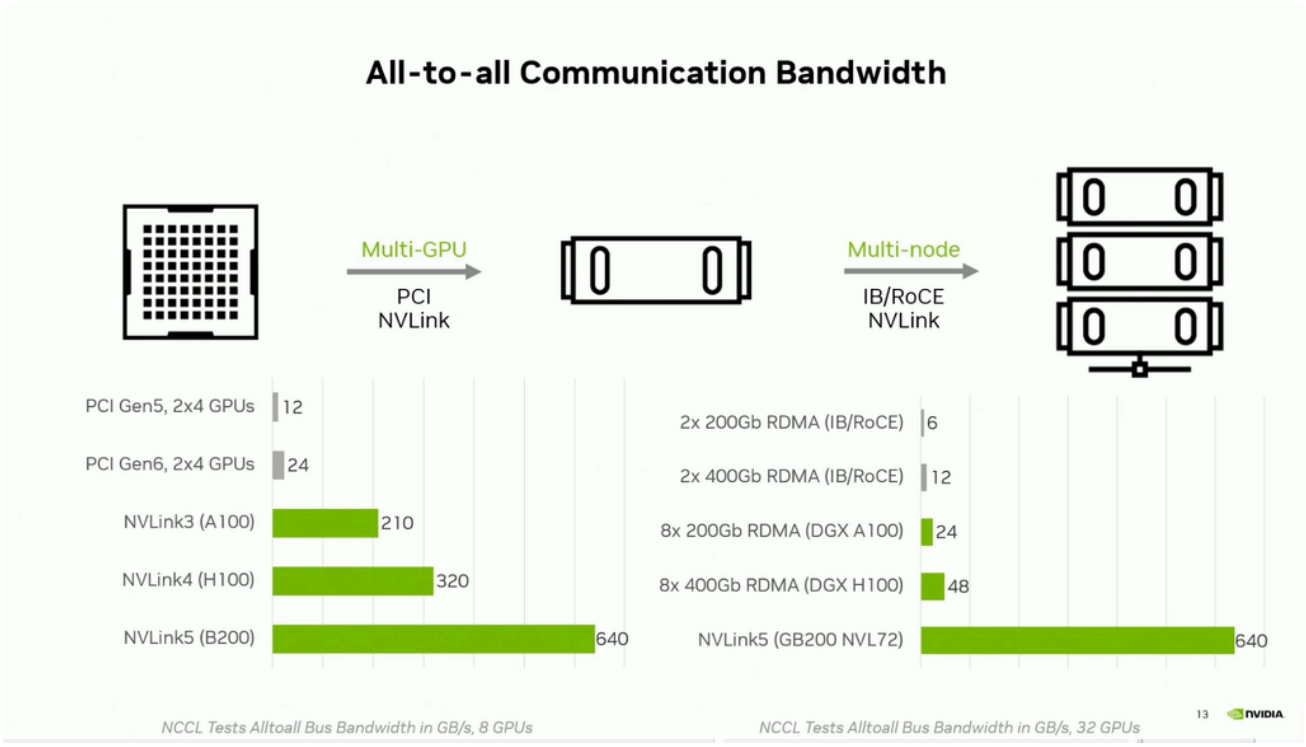
ConnectX-DDR: 2.4 GHz Quad-core (Nehalem) Intel with IB and 10GE switches

(source)

That last plot is from 2011, and the former ones are from 2024 - comparing these you can appreciate how much faster the networks have become and how much bigger messages are being sent.

Here are 2025 performance plots that show the actual achievable bandwidth with the modern technologies in the context of all-reduce and all-to-all collectives:





source

Another tool for bandwidth measurements on NVIDIA GPUs is NVIDIA/nvbandwidth.

Inter-node speed depends on intra-node speed

The specs make inter-node networking look hopeless. On a P6-B200 node (AWS) each accelerator has 900GBps of NVLink 5 but its EFA v4 inter-node is only 50GBps, so the links really are about 18x apart. It is natural to conclude that an `all-reduce` becomes ~18x slower the moment it crosses a node boundary. It doesn't, and this is probably the single most confusing

thing about inter-node networking, so it's worth working through carefully.

footnote: the spec for IB NDR400 for this type of a node is 50GBps per accelerator as well.

The following table aggregates the `busbw` measurements for `all_reduce` measured with all_reduce_bench.py on 1x and 4x P6-B200 nodes - 8x B200 per node, NVLink 5 inside, 8x 50GBps EFA v4 out - on `torch=2.9.1+cu130, cuda=13.0, nccl=2.27.7`. The last column is the price of leaving the node:

payload	1 node	4 nodes	slowdown
32KiB	1.20GBps	0.01GBps	120.0x
64KiB	2.17GBps	0.04GBps	54.2x
128KiB	4.86GBps	0.45GBps	10.8x
256KiB	9.54GBps	1.33GBps	7.2x
512KiB	18.71GBps	2.84GBps	6.6x
1MiB	36.06GBps	5.37GBps	6.7x
2MiB	63.09GBps	10.38GBps	6.1x
4MiB	76.48GBps	18.90GBps	4.0x
8MiB	126.49GBps	35.12GBps	3.6x
16MiB	254.96GBps	64.43GBps	4.0x
32MiB	325.97GBps	91.19GBps	3.6x
64MiB	400.60GBps	156.74GBps	2.6x
128MiB	568.38GBps	197.94GBps	2.9x
256MiB	646.11GBps	229.09GBps	2.8x
512MiB	688.99GBps	326.90GBps	2.1x
1GiB	723.34GBps	361.99GBps	2.0x
2GiB	734.97GBps	372.42GBps	2.0x
4GiB	740.64GBps	377.34GBps	2.0x
8GiB	839.07GBps	380.39GBps	2.2x
16GiB	845.67GBps	381.80GBps	2.2x

At large payloads leaving the node costs about 2x, not 18x.

The reason is that an inter-node NCCL collective does not stop using NVLink - it leans on it for nearly all of the data movement. NCCL reduces within each node first, sends only the reduced shard out over the network, and then broadcasts the result back inside each node. So the slow links carry a small fraction of the bytes, and every NIC in the node is busy at the same time.

Let's do the arithmetic for a 4GiB `all-reduce` over 32 ranks (4 nodes x 8 accelerators). Write `P` for the payload, `g` for the accelerators per node, `k` for the nodes and `n = g*k` for the total ranks - here `P` = 4GiB, `g` = 8, `k` = 4 and `n` = 32. The table reports `busbw`, which is the payload-over-elapsed-time rate scaled by the `all-reduce` correction factor `2*(n-1)/n`, so undoing that

scaling recovers an elapsed time - `P / (busbw / (2*(n-1)/n))`, which for 4 nodes is `4GiB / (377.34GBps / 1.9375)` = 22.05ms, against `4GiB / (740.64GBps / 1.75)` = 10.15ms on a single node, where `n` = 8 makes the factor `2*(8-1)/8` = 1.75. Three ways one might model the 4-node figure:

footnote: mind the bases when doing this yourself - the benchmark prints `1GiB = 2**30 Bytes` but `1GBps = 10**9 Bytes per second`, so 4GiB is 4.29e9 bytes, not 4e9. Dividing GiB by GBps as if they shared a base understates every time in this section by about 7%.

1. the naive inter-node model - the whole payload has to cross one accelerator's NIC: `P / 50GBps` = `4GiB / 50GBps` = 85.9ms. That is 3.9x more than measured, and it is the arithmetic to avoid.
2. a flat ring across all `n` ranks - each link carries `2*(n-1)/n * P` = `2*(32-1)/32 * 4GiB` = 7.75GiB, and a ring laid out over `k` nodes crosses a node boundary `k` = 4 times, so each node has one boundary link's worth to push out. What that costs depends on how many NICs carry it. Over a single accelerator's NIC it would be `7.75GiB / 50GBps` = 166.4ms, 7.5x more than measured - but NCCL opens multiple channels, so different rings cross the boundary at different accelerators and the node's NICs are all in flight, exactly as in model 3 below. Over the node's `g * 50GBps` = 400GBps that is `7.75GiB / 400GBps` = 20.8ms, within 6% of the 22.05ms measured. So this model is *not* ruled out by these numbers.
3. the hierarchical model - reduce inside each node, exchange the shard between nodes, broadcast back inside. Each phase moves its own collective's correction factor times the payload that phase operates on:
    - intra-node reduce-scatter: `(g-1)/g * P` = `(8-1)/8 * 4GiB` = 3.5GiB per accelerator over NVLink
    - inter-node all-reduce of the resulting `P/g` = 0.5GiB shard, across the `k` nodes: `2*(k-1)/k * P/g` = `2*(4-1)/4 * 0.5GiB` = 0.75GiB per accelerator over EFA
    - intra-node all-gather: `(g-1)/g * P` = `(8-1)/8 * 4GiB` = 3.5GiB per accelerator over NVLink

    Per accelerator that is `2*(g-1)/g * P` = 7GiB over NVLink against `2*(k-1)/k * P/g` = 0.75GiB over EFA, so only `0.75/(7+0.75)` = 9.7% of the traffic leaves the node. And because each accelerator drives its own NIC, all `g` NICs are in flight at once, giving the node `g * 50GBps` = 400GBps of inter-node bandwidth rather than one link's 50GBps.

    At wire rate that shard would cross in `0.75GiB / 50GBps` = 16.1ms, against 22.05ms measured - so the exchange effectively runs at ~73% of wire rate, with NIC efficiency and any non-overlapping intra-node time both folded into that figure. The two cannot simply be additive: `16.1 + 10.15` = 26.25ms would exceed the measured time, so the intra-node phases substantially overlap the exchange rather than queueing behind it. So this model brackets the measurement - 16.1ms fully overlapped, 26.25ms fully serialized, with 22.05ms in between.

Models 2 and 3 both land within ~10% of the measurement, so the arithmetic alone does not tell you which one ran - only that the naive model 1 is badly wrong. Which of the two it is depends on the algorithm NCCL selected, and that is directly readable: `NCCL_DEBUG=INFO NCCL_DEBUG_SUBSYS=INIT,TUNING` reports it, and forcing `NCCL_ALGO=allreduce:<algo>` and comparing against the default confirms it. Measured that way on a 4-node 8x H200 `p5en.48xlarge` cluster, the selection was `Ring`. Do not carry that over to the B200 rows above: on that cluster the chooser was AWS's `NCCL_TUNER_PLUGIN=ofi`, and its own log line - `base Tuner is chosen for platform: p5en.48xlarge` - shows it keying off the instance type. A different instance gets a different tuner table, so the only way to know what your own runs pick is to read it.

Either way, two effects rescue the inter-node case: the node's whole NIC bandwidth is used instead of a single link's, and - if the collective is hierarchical - only a fraction of the payload crosses the slow links at all. The first applies whichever model holds, which is why the naive single-NIC arithmetic is wrong by 3.9x before any question of algorithm arises. This is why a fast intra-node fabric matters even for workloads whose bottleneck you would call inter-node - degrade NVLink and the inter-node numbers degrade with it.

Two warnings come with this:

- It is a large-payload effect. The `slowdown` column collapses at small payloads - 120x at 32KiB - because

latency, not bandwidth, dominates there and every node hop adds some. This is the strongest possible argument for bucketing gradients into large reductions rather than reducing each tensor separately.

- Do not read the 4-node `busbw` as a wire speed. 381.80GBps at 16GiB is not what the NICs are doing - undoing the correction factor gives 87.2ms for that reduction, over which each accelerator's NIC moves `1.5 * 16GiB/8` = 3GiB, i.e. ~37GBps, or 74% of its 50GBps. `busbw` is derived from the payload and the elapsed time with a per-collective correction factor, so once NCCL uses a hierarchical algorithm it no longer maps onto any single link - the same caveat as in SHARP.

So what should you expect?

If `busbw` isn't a wire speed, what number can you hold against the NIC's spec? Undo both scalings at once. The elapsed time is `P * (2*(n-1)/n) / busbw`, and during it each accelerator's NICs move `2*(k-1)/k * P/g` bytes, so dividing the bytes by the time cancels the payload entirely - units and all - and leaves a payload-free conversion: `per-accelerator rate = busbw * (k-1)/(n-1)`. For the 4GiB 4-node row that is `377.34GBps * 3/31` = 36.52GBps per accelerator, or 73% of the 50GBps of inter-node bandwidth each accelerator has on this node type - which sits right next to the single-node column's own `740.64GBps / 900GBps` = 82% against NVLink 5. Expressed per accelerator, the inter-node result stops looking anomalous and lands in the same ballpark that intra-node measurements do.

footnote: per accelerator, not per NIC, because the two only coincide when the node puts one interface on each accelerator - as P6-B200 does with its 8 EFA devices for 8 accelerators. P6-B300 puts 16 devices on 8 accelerators, so there the same conversion has to be compared against two interfaces' worth of bandwidth.

footnote: this is the one place in this section where the `GiB` vs `GBps` base trap does not bite, and it's worth seeing why. `P` appears in both the byte count and the elapsed time, so it cancels whatever unit it was quoted in, and the result comes out in `busbw`'s own decimal `GBps` - the same base the interface spec uses, since 400Gbps is 50GBps decimal. Going the long way round does need the conversion, and agrees: the 16GiB warning above reaches ~37GBps by converting 3GiB to 3.22e9 bytes and dividing by 87.2ms, while `381.80GBps * 3/31` = 36.95GBps.

`(k-1)/(n-1)` = `3/31` is the same 9.7% derived above as the share of traffic that leaves the node, which is the tidiest statement of this whole section: `busbw` overstates the wire by exactly the reciprocal of the fraction of bytes that cross it.

Two things to keep in mind about the resulting number. It is comparable to a spec figure, whereas `busbw` is not comparable to anything. And it is a floor rather than a wire measurement, because it charges the whole elapsed time to the NICs even though the intra-node phases overlap the exchange - the true wire rate is somewhat higher than what comes out.

It also only reaches its plateau at large payloads. The same 4-node measurements converted, sorted by payload ascending:

payload	4-node `busbw` GBps	per-accel. GBps	% of spec
16MiB	64.43	6.24	12.5%
32MiB	91.19	8.82	17.6%
64MiB	156.74	15.17	30.3%
128MiB	197.94	19.16	38.3%
256MiB	229.09	22.17	44.3%
512MiB	326.90	31.64	63.3%
1GiB	361.99	35.03	70.1%
2GiB	372.42	36.04	72.1%

payload	4-node `busbw` GBps	per-accel. GBps	% of spec
4GiB	377.34	36.52	73.0%
8GiB	380.39	36.81	73.6%
16GiB	381.80	36.95	73.9%

`% of spec` is against the 50GBps of inter-node bandwidth per accelerator on this node type. Below about 1GiB it falls off a cliff, and at 16MiB a perfectly healthy fabric reports 12.5% - which is worth remembering before reading anything into a small-payload number.

Now the caveat that matters most: these percentages describe this system, and they may or may not translate to another one. The conversion itself is topology only - `k` nodes, `n` ranks, no link speeds appear in it - so that part transfers anywhere. The *value* it produces does not, because it depends on how the intra-node and inter-node fabrics are balanced against each other, and that balance is a property of the node type. B300 shows how far it can shift without even changing accelerator generation: it keeps NVLink 5 at the same 900GBps per accelerator as B200, while its inter-node side is twice as fast - 800Gbps per accelerator against B200's 400Gbps. In the hierarchical model the inter-node term shrinks in proportion while the intra-node terms don't move at all, so a larger share of the elapsed time is intra-node work, and a figure that charges all of that time to the NICs has to be read differently. What it actually comes out as there is unknown - this benchmark has not been run on a B300 cluster - so derive the number from that system's own measurements instead of carrying 73% across.

footnote: this conversion assumes NCCL used the hierarchical algorithm - model 3 in Inter-node speed depends on intra-node speed. Under a flat ring each link would instead carry `2*(n-1)/n * P` across `k` node boundaries and the conversion would not apply, so capture `NCCL_DEBUG=INFO NCCL_DEBUG_SUBSYS=INIT,TUNING` alongside the numbers when the algorithm isn't already known. That the numbers land where a hierarchical algorithm predicts is evidence for it, not proof of it.

For how to get these numbers out of a provider in the first place, see Ask for the actual performance numbers.

Measuring the inter-node fabric on its own

A collective can't tell you this. An `all-reduce` across nodes will always lean on the intra-node fabric, which is the whole point of this section. To measure the wire itself, use a point-to-point RDMA benchmark. `ib_write_bw` from perftest runs between two hosts using one adapter (`-d`) and one queue pair (`-q`, default 1), so no second local accelerator and no collective are involved - the isolation is the tool's construction rather than something you configure. Add `--use_cuda=<gpu>` for the GPUDirect RDMA path out of accelerator memory instead of host memory, and `-a` to sweep payload sizes for comparison against the `busbw` table.

Despite the name it is not InfiniBand-only - it is written over `uverbs`, the userspace RDMA API, so it works on any adapter the RDMA stack enumerates. On InfiniBand a Subnet Manager must be running first. On EFA pass `-c SRD`, since the default RC connection type doesn't exist there - omit it and the run dies at setup with `Unable to create QP` / `Couldn't create IB resources`, because there is no RC transport to fall back to. Both sides need identical options and identical `perftest` versions, and the result is a synthetic operation stream rather than application traffic.

Fabrics with their own userspace stack rather than verbs - Slingshot, Omni-Path via OPX, GPUDirect-TCPX - need their own tools. The nearest equivalent there is `fi_rma_bw` from libfabric's fabtests, an RMA read and write bandwidth test which is the same shape of measurement as `ib_write_bw`. Note that libfabric's own documentation warns that its tests are "designed to test functionality and not performance", with the benchmarks as the stated exception, and that even those are "not guaranteed to provide the best latency or bandwidth performance numbers a given provider or system may achieve" - so treat the figure as a floor rather than the fabric's capability.

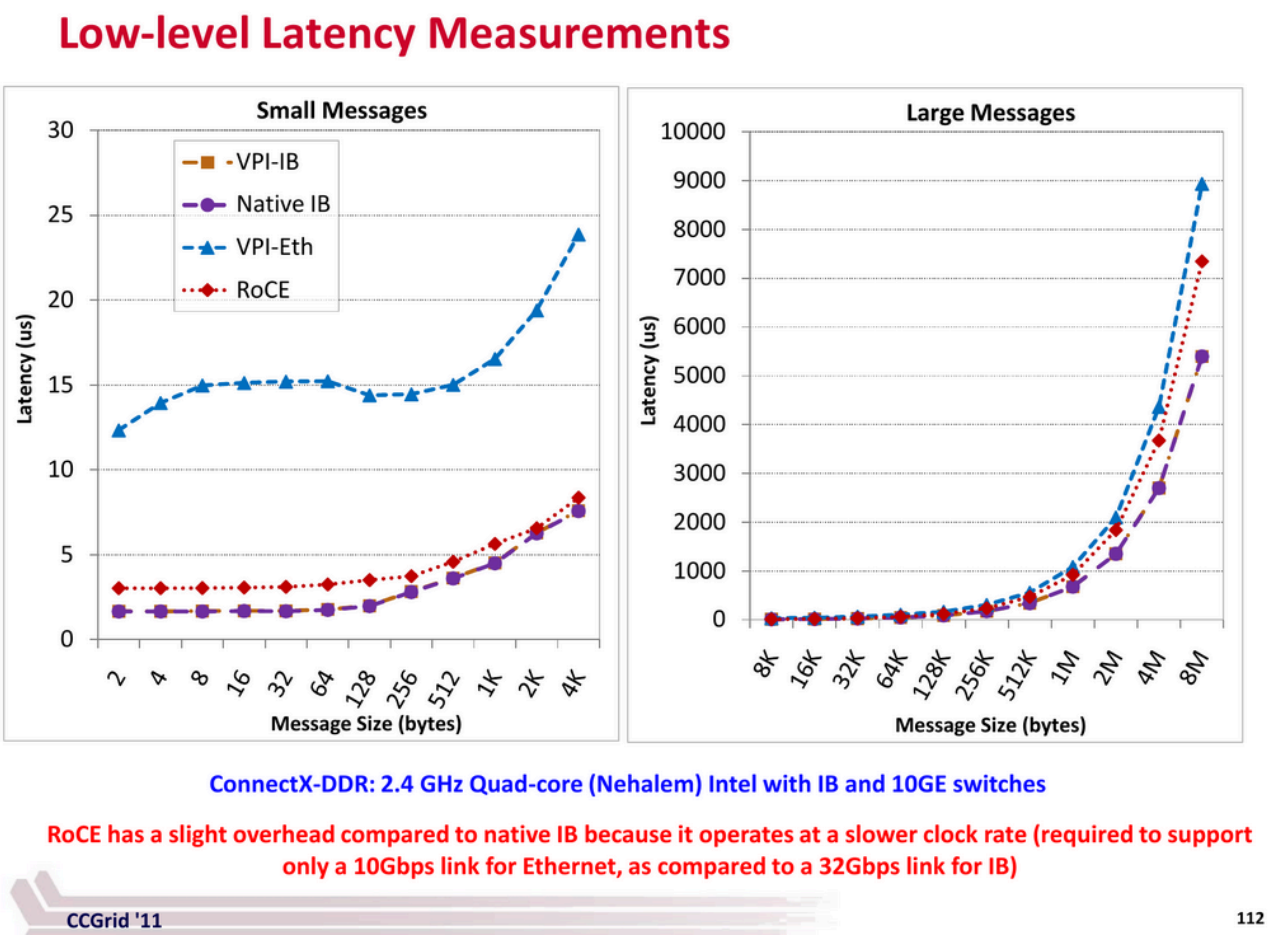
footnote: RDMA-write-over-SRD support was contributed to `perftest` by AWS (PR 206). Confirmed on a live instance on 2026-08-07: on an 8x H200 `p5en.48xlarge` node, `ib_write_bw -d rdmap110s0 -c SRD -x 0 --report_gbits -D 10` between two nodes sustained **193.72Gbps on one adapter, or 96.9% of that adapter's 200Gbps line rate.**

footnote: it is tempting to instead keep all the accelerators and take the intra-node fabric away with `NCCL_P2P_DISABLE=1`, but that only pushes intra-node traffic down to host memory - the documented fallback order is P2P, then SHM, then network, so `NCCL_SHM_DISABLE=1` is needed as well before NCCL puts same-node ranks on a NIC. On EFA even that isn't enough, because the libfabric provider uses the instance's shared memory for intra-node communication: NCCL asks for the network and libfabric serves the transfer from host memory instead. `FI_EFA_ENABLE_SHM_TRANSFER=0` overrides that.

Latency

Latency tells us how long it takes to send or receive a message. It has an inverse relationship with throughput - the faster is the throughput the lower is the latency.

Here is an old but good plot demonstrating how the latencies change with message size and the type of the network:



(source)

Typically the more "hops" the message has to travel, the bigger the latency. 2 accelerators residing on the same node and connected directly to each other (e.g., NVLink) will have the least amount of latency. If their communication path traverses a PCIe switch the latency will be bigger. 2 accelerators residing on 2 different nodes sharing a single switch will have a bigger latency because there is a switch to traverse. The further they get away from each other, the more switches the message has to travel through, the bigger the latency.

Proprietary network hardware and NCCL

Proprietary network hardware vendors like AWS (EFA) don't disclose their secrets and therefore the public libraries like nccl cannot support those out of the box. These vendors have to supply their own versions of the network collective libraries to be used by users of their hardware.

Originally proprietary hardware vendors used the trick of telling the users to use `LD_LIBRARY_PATH` and/or `LD_PRELOAD` to dynamically overload `libnccl.so` to get their custom version loaded into PyTorch or another framework. But recently NCCL developed a NCCL Net Plugin which should be used now instead. This feature was added in NCCL v2.12.

Now, when NCCL is initialized, it will look for a `libnccl-net.so` library and dynamically load it, then look for symbols inside the library. That's where proprietary hardware vendors should now put their custom APIs. This library, of course, should still be either in `LD_LIBRARY_PATH` or the `/etc/ld.so.conf` config.

For more information about dynamic library loading see this section.

Node Proximity

If you get 2 random nodes from the cloud they may not reside on the same subnet and there will be an additional latency incurred for all transmissions.

You want to make sure that the nodes used for a single training all reside on the same subnet/spine so they are all one hop away from each other.

When you plan to eventually have a large cluster but starting small make sure that your provider can expand the cluster while keeping all the nodes close to each other.

Here are the cloud-specific ways of accomplishing node proximity:

- Azure: availability set
- AWS: cluster placement groups
- GCP: compact placement policies

Depending on the type of package you have or what type of machines you rent - you may or may not be able to use those.

Shared internode network

If you use a shared HPC environment, or even if you have your own cluster but sharing it with your colleagues expect the network bandwidth to be unreliable and fluctuate at different times of the day.

This situation unfortunately makes it extremely difficult to finetune the performance of your training setup. Since every time you run a test the TFLOPS will vary, so how do you do the optimization? This is at least the situation with SLURM-based clusters. Apparently when Kubernetes is used, one can use cluster namespaces to segregate the network.

case study: we had this issue at JeanZay HPC when we were doing preliminary experiments before we started training BLOOM-176B. As that HPC has many users it was pretty much impossible to do speed optimizations, as even running the exact same setup again and again gave different throughput results. Luckily just before we launched BLOOM-176B training we were given an exclusive access to the new at that time A100 partition so we were the only users and we were able to greatly optimize the throughput.

Parallelism network collectives

See Parallelism network collectives.

Communication Patterns

The intention of this chapter is not to show code examples and explain APIs for which there are many tutorials, but to have excellent visuals that explain how the various types of communication patterns work.

Point-to-point communications

Point-to-point communications are the simplest type of communication where there is always a single sender and a single receiver.

For example, Pipeline Parallelism performs a point-to-point communication where the activations from the current vertical stage is sent to the next stage. So the current gpu performs `send` and the gpu holding the next stage performs `recv`.

PyTorch has `send` and `recv` for blocking, `isend` and `irecv` for non-blocking p2p comms. more.

One-sided communications

Collectives and `send`/`recv` are two-sided: every rank that participates posts a matching call. One-sided (RMA) communication lets one rank write into or read from another rank's memory without that peer posting `recv`. That is the useful shape for irregular access - an embedding lookup, a weight copy, routing a token to one expert - where making every rank join an all-to-all would move more data than the algorithm needs.

Starting from PyTorch 2.14, `nccl2` exposes this as a *window* over the NCCL `Put`/`Get` APIs. Creating the window is still collective - every rank in the group must call it in the same order - and as of 2.14 the Python entry points are experimental (`dist._new_window`, `dist._supports_window`). Use it when the access pattern is sparse; keep two-sided collectives when every rank participates.

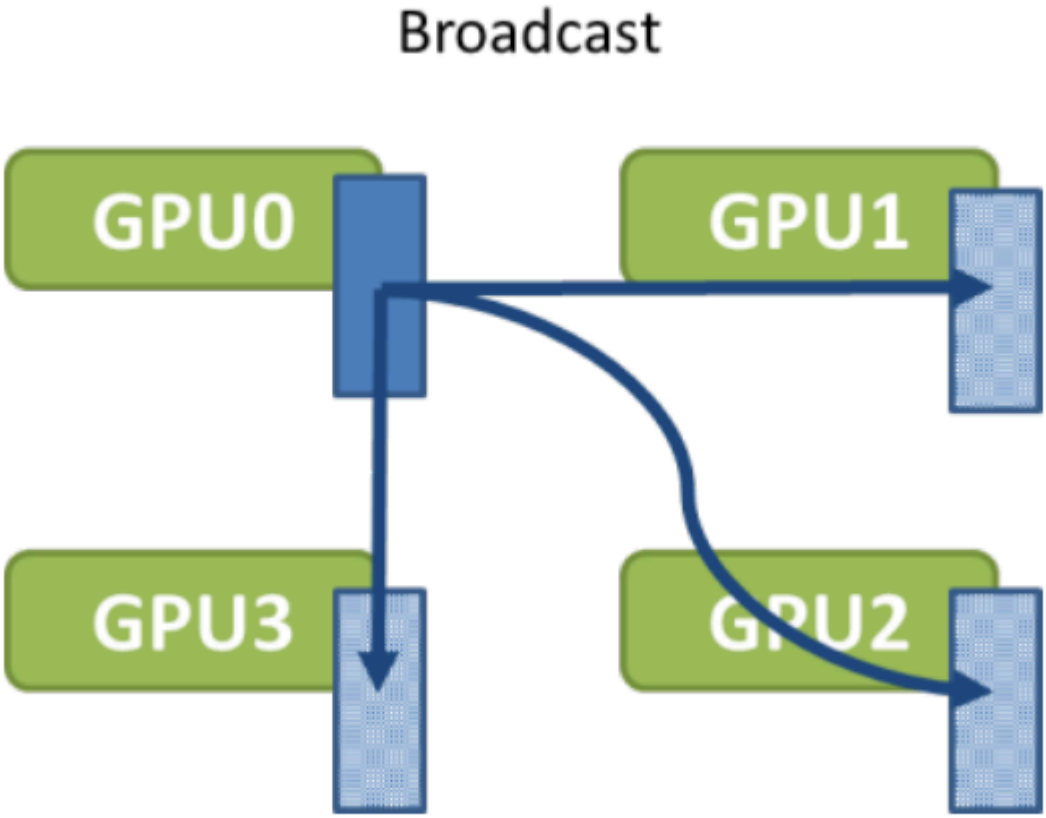
Reference: PyTorch 2.14 release blog.

Collective communications

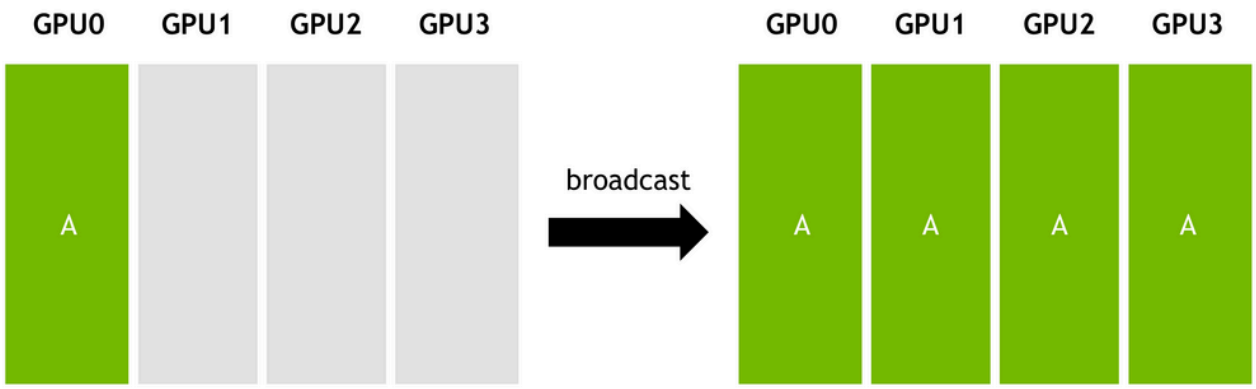
Collective communications include either multiple senders and a single receiver, a single sender and multiple receivers or multiple senders and multiple receivers.

In the world of PyTorch typically each process is tied to a single accelerator, and thus accelerators perform collective communications via process groups. The same process may belong to multiple process groups.

Broadcast



source

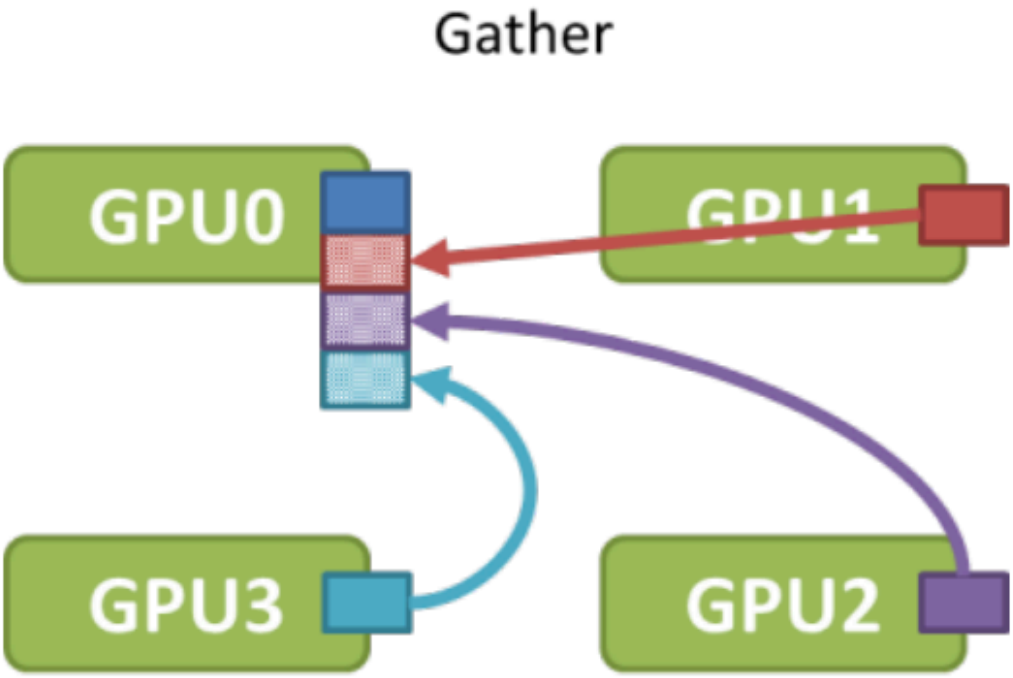


source

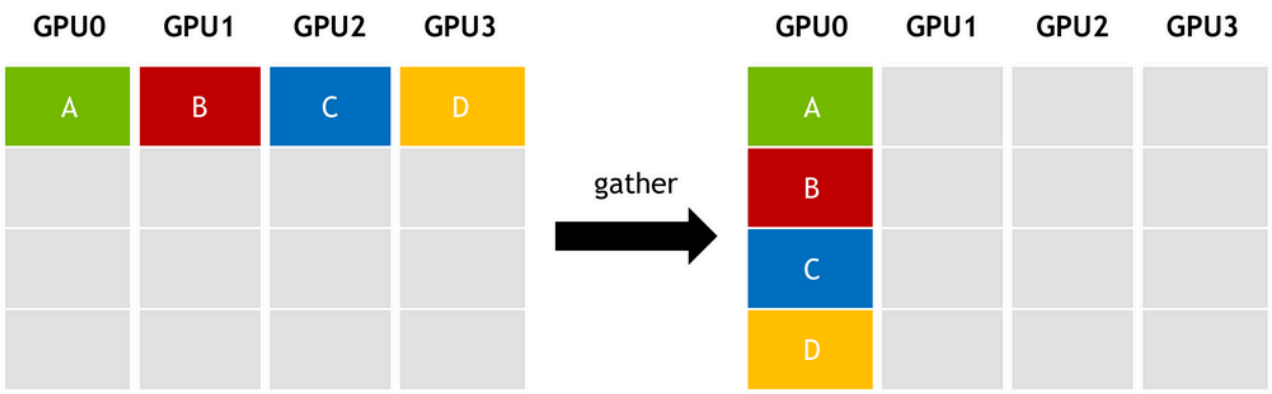
PyTorch API example:

`dist.broadcast(tensor, src, group)`: Copies `tensor` from `src` to all other processes. doc.

Gather



source

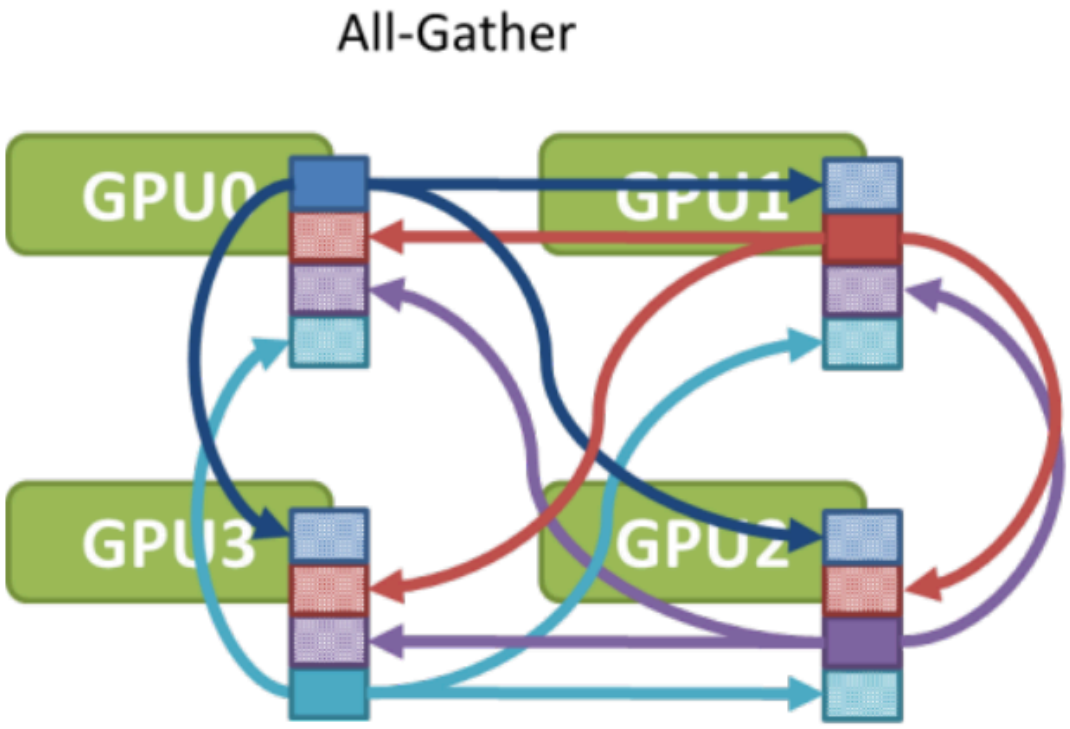


source

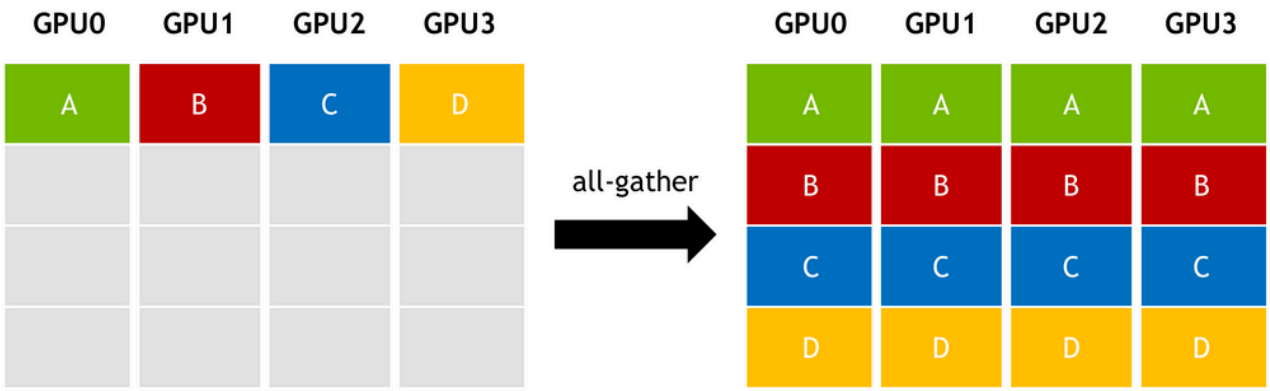
PyTorch API example:

`dist.gather(tensor, gather_list, dst, group)`: Copies one `tensor` from every process in the group into `gather_list` on `dst`. doc

All-gather



source



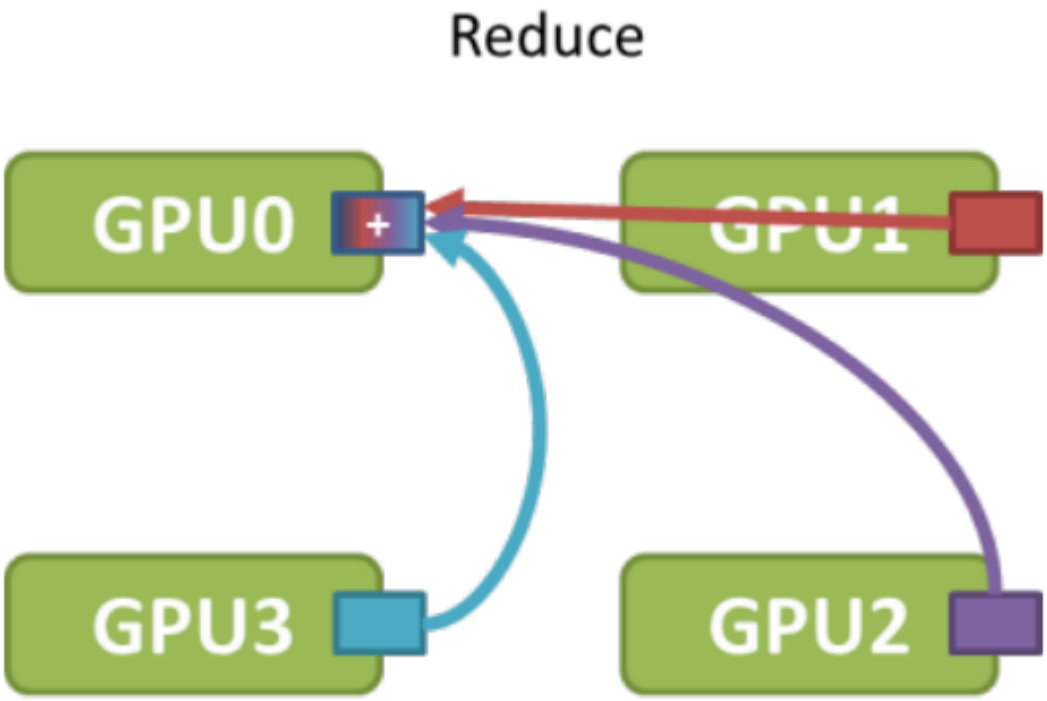
source

For example, this collective is used in ZeRO (DeepSpeed and FSDP) to gather the sharded model weights before `forward` and `backward` calls.

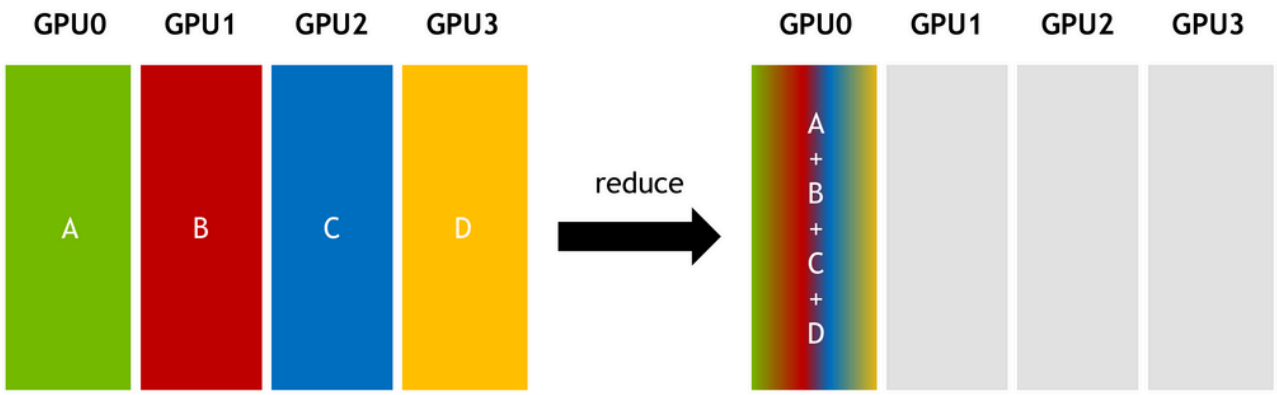
PyTorch API example:

`dist.all_gather(tensor_list, tensor, group)`: Copies `tensor` from all processes to `tensor_list`, on all processes. doc

Reduce



source



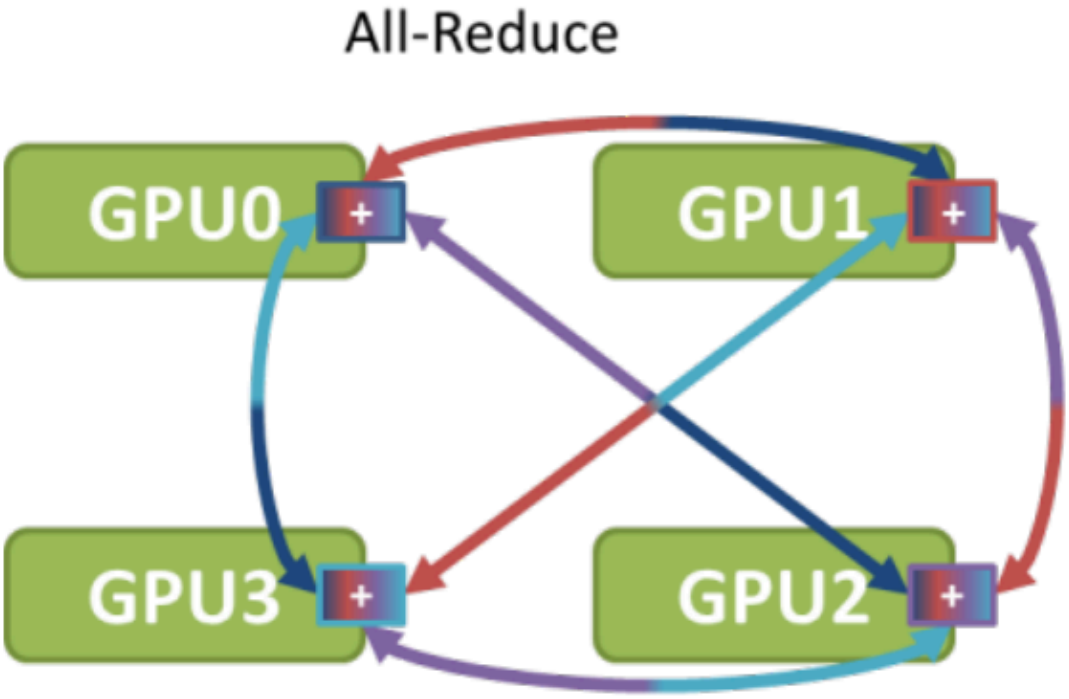
source

PyTorch API example:

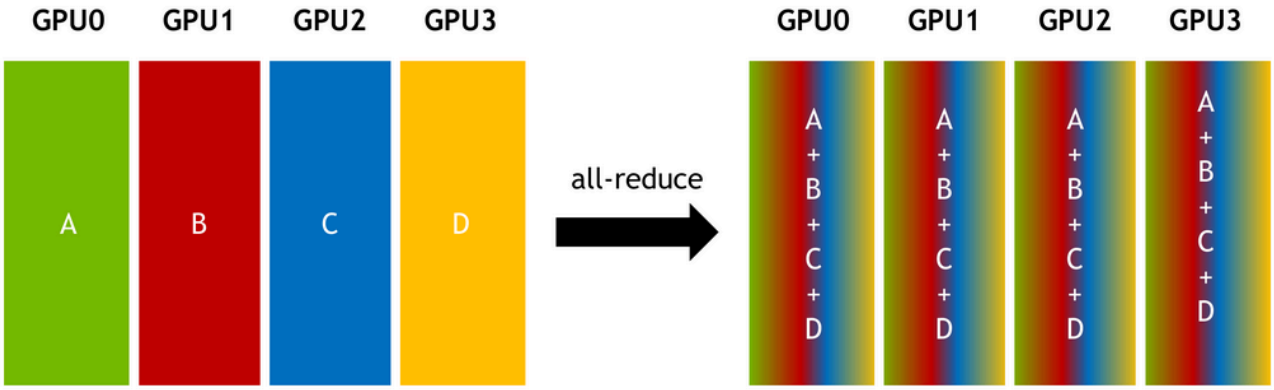
`dist.reduce(tensor, dst, op, group)`: Applies `op` to every `tensor` and stores the result in `dst`. doc

PyTorch supports multiple reduction operations like: `avg`, `sum`, `product`, `min`, `max`, `band`, `bor`, `bxor`, and others - full list.

All-reduce



source



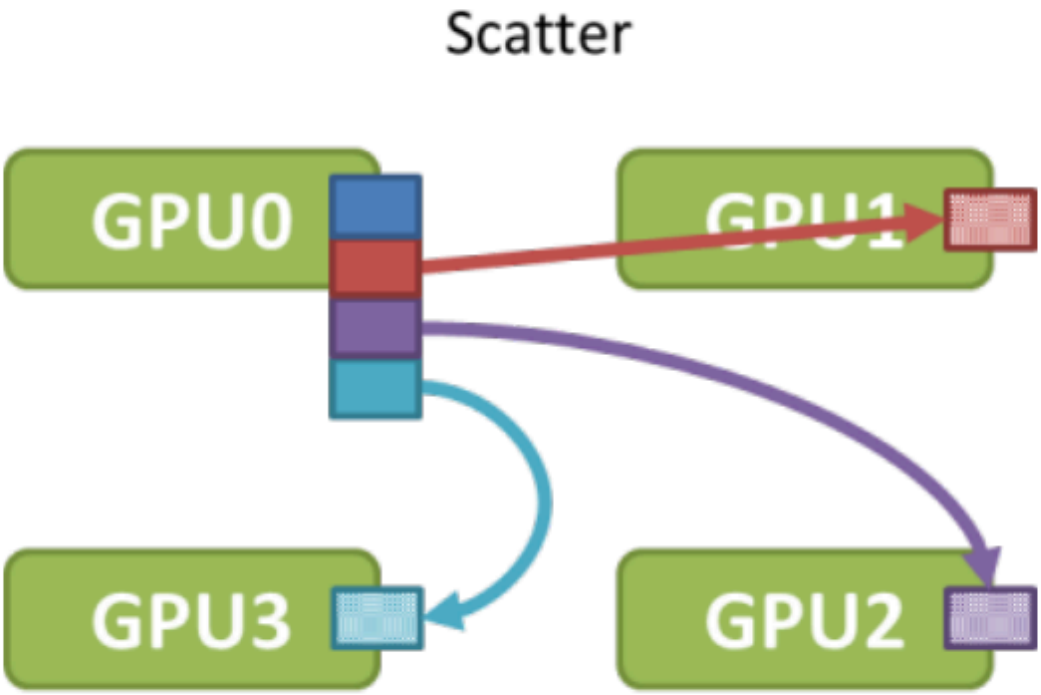
source

For example, this collective is used in DDP to reduce gradients between all participating ranks.

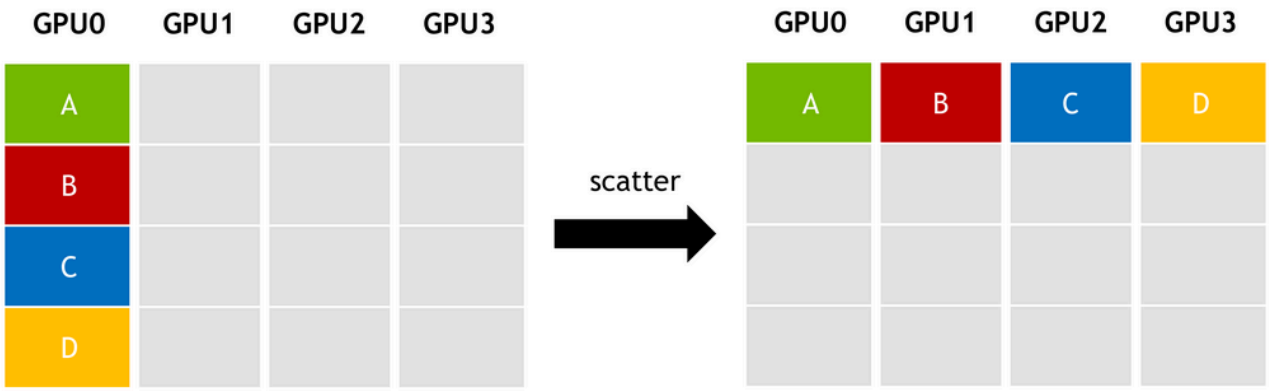
PyTorch API example:

`dist.all_reduce(tensor, op, group)`: Same as reduce, but the result is stored in all processes. doc

Scatter



source

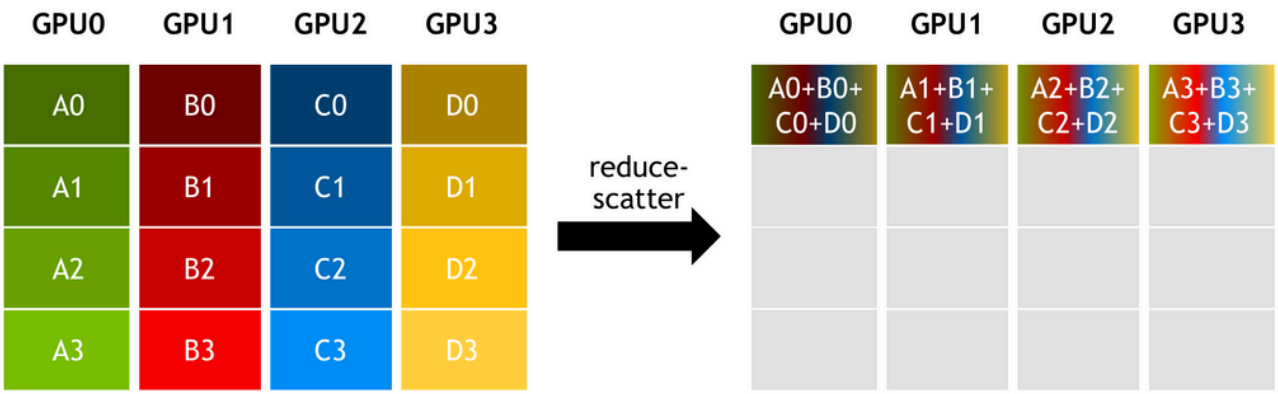


source

PyTorch API example:

`dist.scatter(tensor, scatter_list, src, group)`: Copies the `i`-th tensor `scatter_list[i]` to the `i`-th process. doc

Reduce-Scatter



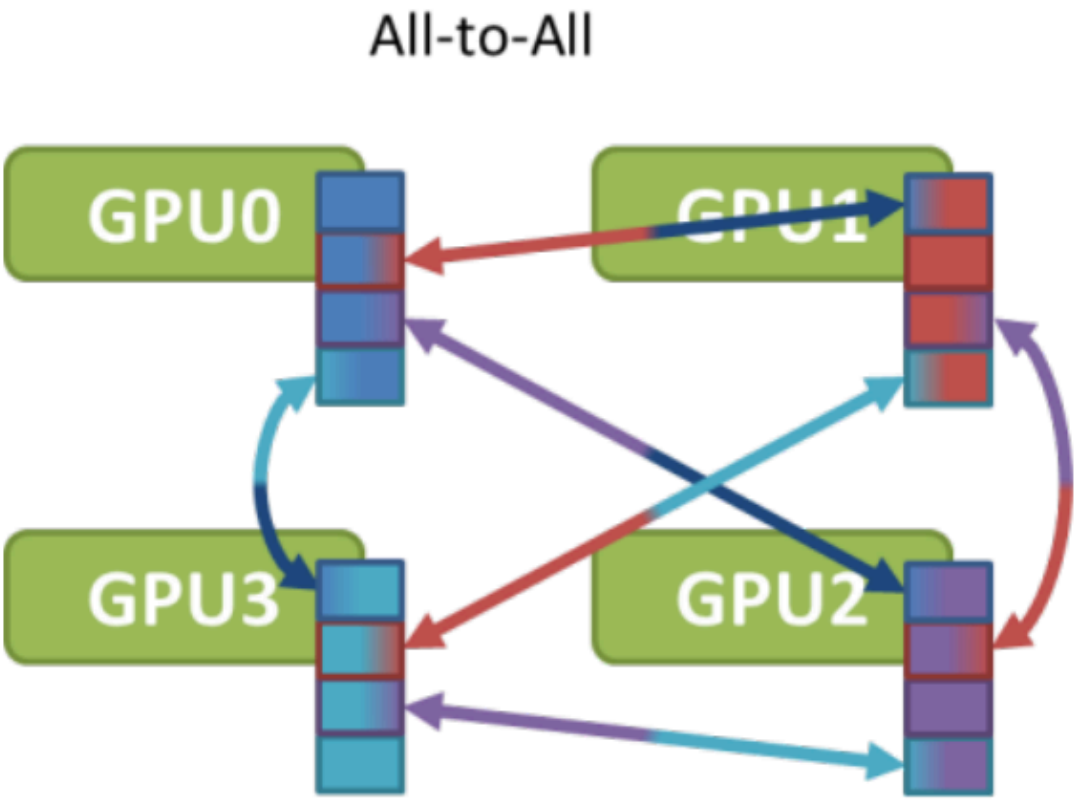
source

For example, this collective is used in ZeRO (DeepSpeed and FSDP) to efficiently reduce gradients across all participating ranks. Note that it isn't a cheaper drop-in for all-reduce - each rank ends up with only its own reduced shard rather than the full result. That is exactly what ZeRO wants, and it's why it costs half the data movement: a ring all-reduce is a reduce-scatter followed by an all-gather, so dropping the all-gather phase halves the bytes on the wire.

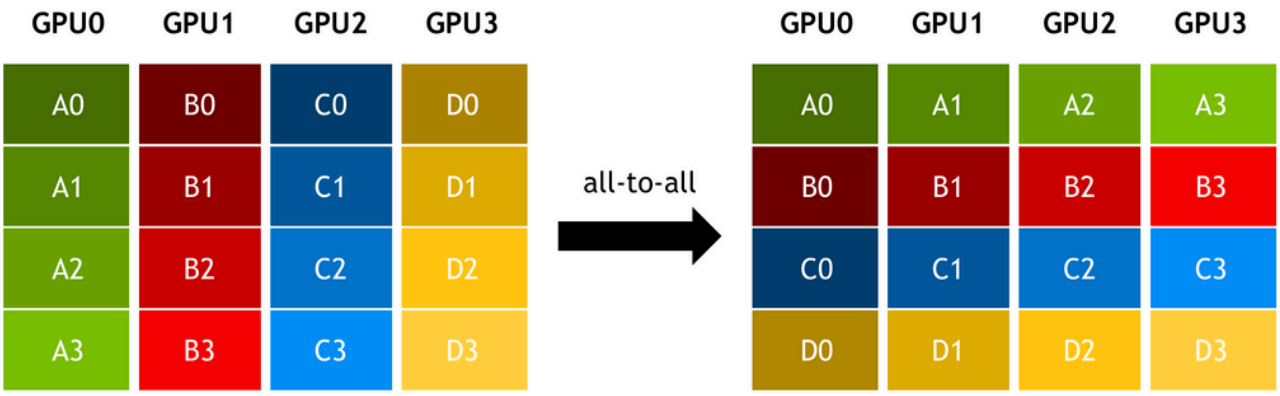
PyTorch API example:

`dist.reduce_scatter(output, input_list, op, group, async_op)`: Reduces, then scatters a list of tensors to all processes in a group. doc

All-to-all



source



source

For example, this collective is used in DeepSpeed Sequence Parallelism for attention computation, and in MoE Expert Parallelism.

PyTorch API example:

`dist.all_to_all(output_tensor_list, input_tensor_list, group)`: Scatters list of input tensors to all processes in a group and return gathered list of tensors in output list. doc

Algorithms

The collective communications may have a variety of different implementations, and comm libraries like `nccl` may switch between different algorithms depending on internal heuristics, unless overridden by users.

Ring

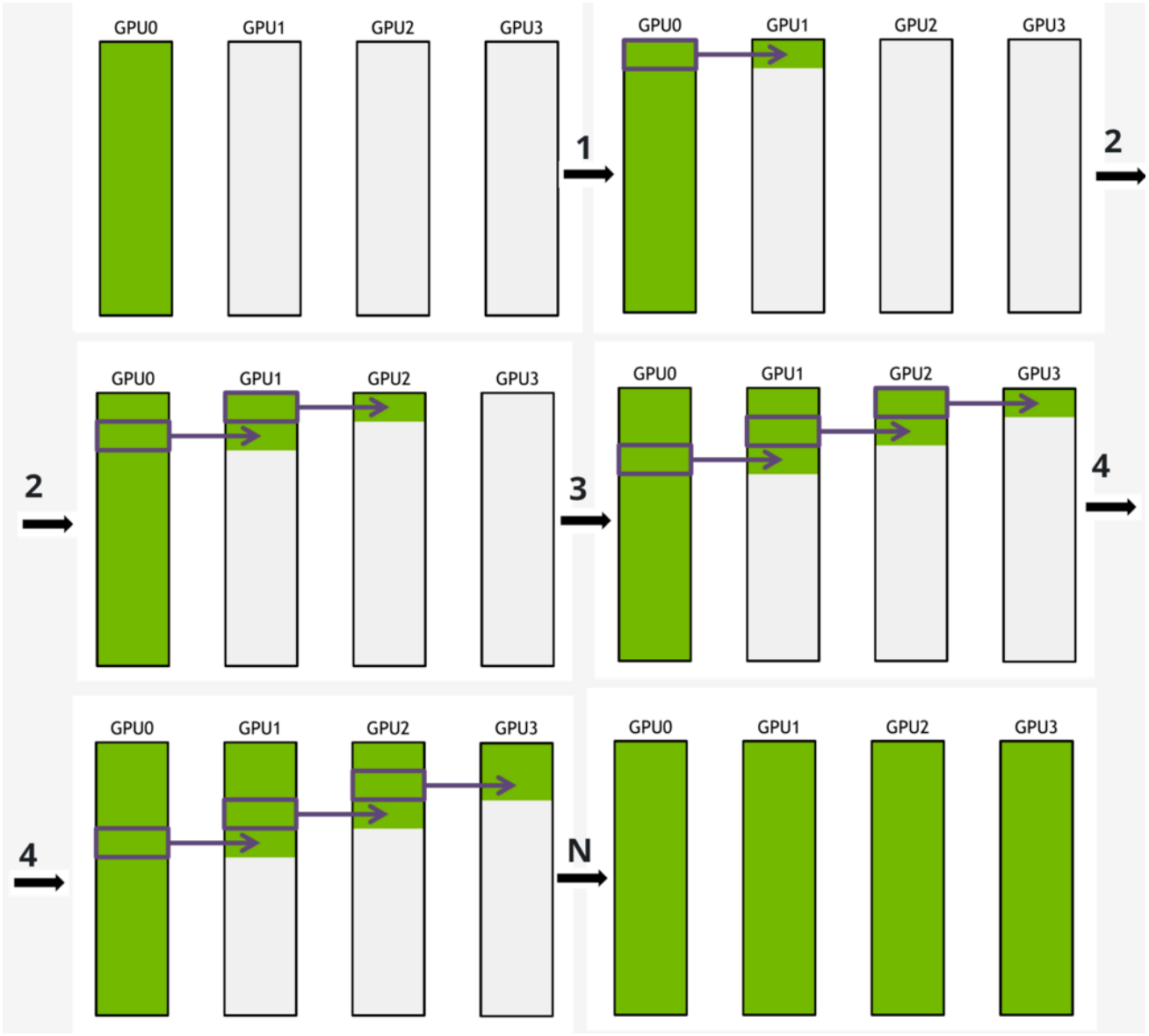
Broadcast with unidirectional ring

Given:

- N: bytes to broadcast
- B: bandwidth of each link
- k: number of GPUs

A naive broadcast will send `N/B` at each step. The total time to broadcast to `k` GPUs will take: `(k-1)*N/B`

Here is an example of how a ring-based broadcast is performed:



source

This algorithm splits `N` into `S` messages

At each step `N/(S*B)` is sent, which is `S` times less than the naive algorithm sends per step.

The total time to broadcast `N` bytes to `k` GPUs will take:

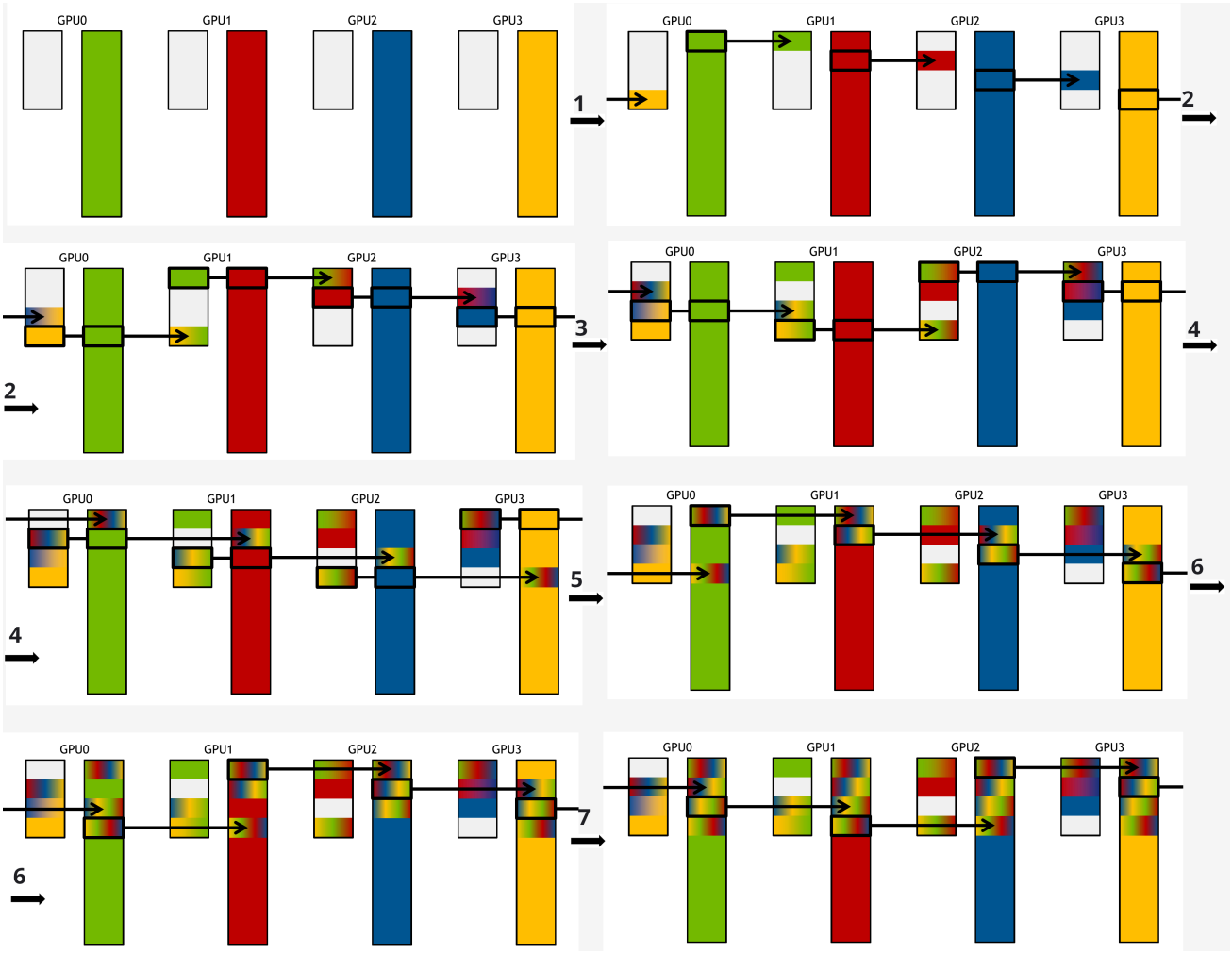
```
S*N/(S*B) + (k - 2)*N/(S*B) = N*(S + k - 2)/(S*B)
```

and if split messages are very small so that `S>>k`: `S + k - 2` is `~S` and then the total time is about `N/B`.

All-reduce with unidirectional ring

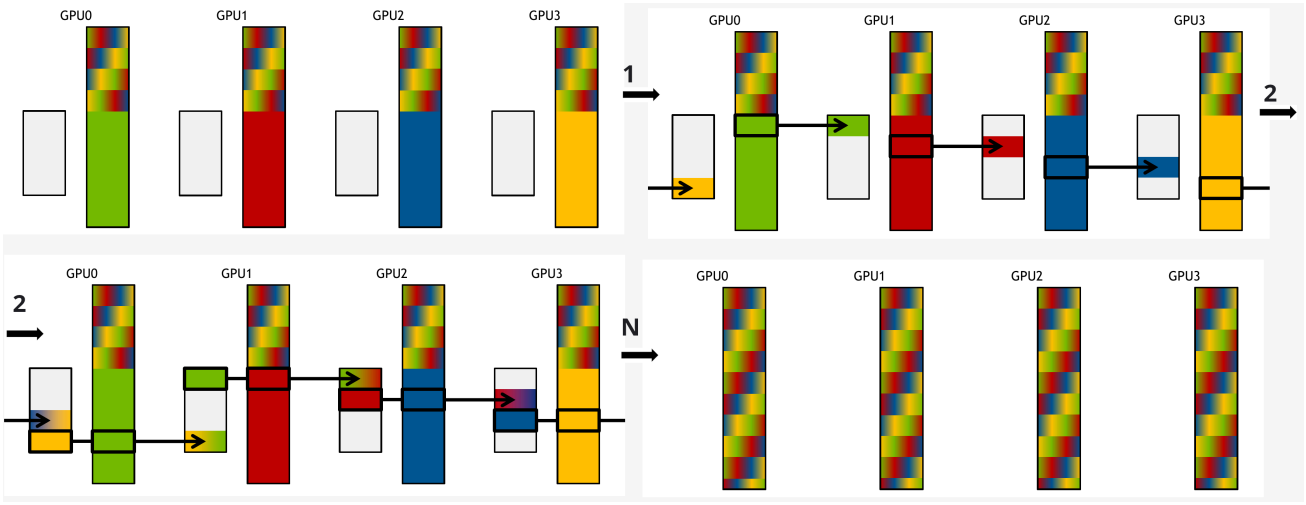
Ring-based `all-reduce` is done similarly to broadcast. The message is split into many small messages and each GPU sends a small message to the next GPU in parallel with other GPUs. `all-reduce` has to perform 2x steps than `broadcast`, because it performs a reduction - so the size of the message needs to be sent twice over the wire.

Moreover, the whole message can be first split into chunks, to make the process even more efficient. Here is the reduction of the first chunk:



source

Then the next chunk is done, until all smaller messages are reduced:



source

Tree

A ring passes the message from neighbor to neighbor, so the number of steps grows with the number of GPUs `k`, and the base latency grows with it. A tree fans the message out instead, so the hop count grows with `log(k)`. That is why the two swap places as you scale: ring keeps the higher peak bandwidth, tree the lower base latency, and for a fixed message size a large enough `k` makes tree the faster of the two.

A single binary tree would waste half the machines: half the nodes are leaves, and a leaf only ever receives, so its outbound bandwidth never contributes. NCCL therefore builds a *double binary tree* - two trees over the same GPUs, arranged so no node is an interior node in both and at most one is a leaf in both. Both carry traffic simultaneously, which recovers the full bandwidth. The second tree is the first mirrored when the node count is even, or shifted by one position when it is odd.

For how NCCL picks between ring and tree, and how to see or override the choice, see `NCCL_ALGO`.

Protocols

The algorithm decides who sends what to whom. Orthogonal to it, the protocol decides how each of those transfers is synchronized, and it swings performance as much as the algorithm does. NCCL has three of them - this is the `proto` half of the `Algo NVLS proto SIMPLE` lines that `NCCL_ALGO` shows you.

Columns are ordered by ascending per-hop latency, which happens to be ascending bandwidth utilization too:

Property	LL	LL128	Simple
Design goal	low latency	low latency and high bandwidth	high bandwidth
Synchronization	flag-based	flag-based	memory fences
Payload per unit	4B data + 4B flag	120B data + 8B flag	large data chunks
Bandwidth utilization	25-50% of peak	~95% of peak	near peak
Latency per hop	~1µs	~2µs	~6µs

Sources:

1. Demystifying NCCL: An In-depth Analysis of GPU Communication Protocols and Algorithms (2025), Table I.

`Simple` sends large chunks and uses memory fences to guarantee the receiver sees a complete chunk. That reaches near-peak bandwidth, but the fence cost dominates anything small. `LL` drops the fences: it pairs 4 bytes of data with a 4-byte flag and writes the 8 bytes atomically, so the receiver proceeds the moment the flag lands. The catch is that the staging buffer has to sit in host memory for the CPU to poll that flag, which rules out GPUDirect RDMA and caps `LL` at 25-50% of peak - acceptable when latency is all that matters, wasteful otherwise.

`LL128` keeps the flag trick at 128-byte granularity, 120 bytes of data to an 8-byte flag, which is where the ~95% of peak comes from. It pays with a hardware requirement: those 128-byte writes must never be split or reordered by the interconnect, and where that isn't guaranteed - some PCIe paths - NCCL disables `LL128` rather than risk corruption. So it is what you see on NVLink systems, and its strength is intra-node. Over a network the per-128-byte synchronization turns into millions of small operations on a large transfer, and `Simple` takes the lead again.

More guides

Here are some additional guides with good visuals:

- UvA Deep Learning Tutorials

NCCL-specific:

- Demystifying NCCL: An In-depth Analysis of GPU Communication Protocols and Algorithms (2025) - this paper presents a comprehensive analysis of NCCL, focusing on its communication protocol variants (Simple, LL, and LL128), mechanisms governing intra-node and inter-node data movement, and ring- and tree-based collective communication algorithms.

Network Debug

Often you don't need to be a network engineer to figure out networking issues. Some of the common issues can be resolved by reading the following notes.

Glossary

- Bonding: using multiple NICs together for faster speed or as a back up
- HCA: Host Channel Adapter - InfiniBand's term for a network adapter
- IB: InfiniBand (Originally by Mellanox, acquired by NVIDIA)
- NIC: Network Interface Card
- OOB: Out-of-Band (typically a slower ethernet NIC)
- RDMA: Remote Direct Memory Access

How to diagnose NCCL multi-gpu and multi-node connectivity issues

See also Debugging multi-node training for debugging multi-node issues at the PyTorch level.

This section is definitely non-exhaustive and is meant to cover some of the most common setup issues that I have often encountered. For more complex problems please research the NCCL repo Issues or file a new Issue if you can't find one matching your situation. NCCL also includes a brief troubleshooting section but usually one learns a lot more from reading Issues.

For the network diagnostics work, instead of using a full application which may take a long time to launch and have unrelated issue, I recommend using this specially developed design test script: torch-distributed-gpu-test.py.

First, run the nccl-based program after setting:

```
export NCCL_DEBUG=INFO
```

which will print a lot of debug info about the NCCL setup and its network traffic.

For example if you're using the aforementioned debug script, for a single node with 8 GPUs, you might do:

```
NCCL_DEBUG=INFO torchrun --nproc_per_node 8 --nnodes 1 torch-distributed-gpu-test.py
```

To launch it on multiple nodes, you'd have to either use some orchestration software like SLURM or Kubernetes, or manually launch it on each node (`pdsh` would be of a huge help) - see the instructions inside torch-distributed-gpu-test.py for details. But to understand how things work I recommend starting with just 1 node and then progressing to 2, and later to more nodes.

Now, inspect the output of the program and look for a line that starts with:

```
NCCL INFO NET/
```

and then inspect which protocol and which interfaces it is using.

For example, this output:

```
NCCL INFO NET/FastSocket : Using [0]ibs108:10.0.19.12<0> [1]ibs109:10.0.19.13<0>
[2]ibs110:10.0.19.14<0> [3]ibs111:10.0.19.15<0> [4]ibs112:10.0.19.16<0> [5]ibs113:10.0.19.17<0>
[6]ibs114:10.0.19.18<0> [7]ibs115:10.0.19.19<0>
```

tells us that nccl-fastsocket transport layer plugin is used and it discovered 8 `ibs*` network interfaces (NIC cards). If you're using Google Cloud this is correct, and your NCCL is likely setup correctly. But if you're using InfiniBand (IB) and you're getting the above output, you're likely to clock a very low internode speed, because this means that you activated the wrong plugin.

In the case of IB, what you want to see is `NET/IB` and its IB interfaces:

```
NCCL INFO NET/IB : Using [0]mlx5_0:1/IB [1]mlx5_1:1/IB [2]mlx5_2:1/IB [3]mlx5_3:1/IB [4]mlx5_4:1/
IB [5]mlx5_5:1/IB [6]mlx5_6:1/IB [7]mlx5_7:1/IB [RO]; OOB eno1:101.262.0.9<0>
```

Here, you can see that IB is used with 8 `mlx5_*` interfaces for collective comms, and one OOB, which stands for Out-Of-Band, and is used for doing bootstrapping the connections and is usually using a slower Ethernet NIC (at times several NICs bonded into one - in case you're wondering what does `bond` in the interface name stand for).

To know which TCP/IP interfaces your node has you run `ifconfig` on one of the nodes (typically all similar nodes will have the same interface names, but not always).

If your collective comms network is IB, instead of `ifconfig` you'd run `ibstat`. The last example of `NCCL INFO NET` would correspond to the following output:

```
$ ibstat | grep mlx5
CA 'mlx5_0'
CA 'mlx5_1'
CA 'mlx5_2'
CA 'mlx5_3'
CA 'mlx5_4'
CA 'mlx5_5'
CA 'mlx5_6'
CA 'mlx5_7'
```

Which of these commands answer at all depends on the fabric, so it helps to know what each one asks. `ibstat`, along with `ibnetdiscover`, `iblinkinfo`, `ibhosts`, `ibswitches` and `perfquery`, queries the subnet manager - the fabric-wide service that hands out addresses and answers topology questions on InfiniBand. Those report only on a real InfiniBand fabric, which is what the sample above is. `ibv_devices`, `ibv_devinfo` and `ibstatus` ask the local adapter instead, so they answer on any fabric the RDMA stack enumerates, as does reading the port directly with `cat /sys/class/infiniband/*/ports/1/rate`.

So identify the fabric first, using a command from the second group. On InfiniBand the port's `link_layer` is `InfiniBand` and the subnet-manager tools work. On RoCE, which carries RDMA over Ethernet, `link_layer` is `Ethernet`. On EFA there is no subnet manager: `ibv_devinfo` reports `transport: unspecified` with `link_layer: Unspecified`, `ibstatus` reports each port `ACTIVE` at its rate, and the subnet-manager tools fail - `ibnetdiscover` with `Can't open SMI UMAD port`, while

`ibstat -l` prints nothing at all. That silent empty output is the trap, since it reads as "this node has no RDMA devices"; on a 2026-09 AWS node measured for this chapter it printed nothing while the node had 16 adapters, each `ACTIVE` at `200 Gb/sec (4X HDR)`.

Since besides the fast inter-node connectivity NICs, you're also likely to have a slow management Ethernet NIC (or even several of those), that is there to be able to configure the node, use a shared file system, access the Internet, it's almost certain that `ifconfig` will also include additional NICs. Also you are likely to have a docker network interface, `lo` loopback and some others. For example on my desktop I may get the following output:

```
$ ifconfig
docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.99.0.1  netmask 255.255.0.0  broadcast 172.99.255.255
        inet6 f330::42:fe33:f335:7c94  prefixlen 64  scopeid 0x20<link>
        ether 02:42:fe:15:1c:94  txqueuelen 0  (Ethernet)
        RX packets 219909  bytes 650966314 (650.9 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 262998  bytes 20750134 (20.7 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 1147283113  bytes 138463231270 (138.4 GB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 1147283113  bytes 138463231270 (138.4 GB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.0.23  netmask 255.255.255.0  broadcast 10.0.0.255
        inet6 2601:3108:1c71:600:4224:7e4b:13e4:7b54  prefixlen 64  scopeid 0x0<global>
        ether 04:41:1a:16:17:bd  txqueuelen 1000  (Ethernet)
        RX packets 304675330  bytes 388788486256 (388.7 GB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 74956770  bytes 28501279127 (28.5 GB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
        device memory 0xa3b00000-a3bfffff
```

The reason I mention all these is that the critical part is to ensure that NCCL reports only the correct interfaces in its `Using` debug line. If any interfaces like `docker0` or `lo` or `eth0` end up being reported, e.g.:

```
NCCL INFO NET/Socket : Using [0]eth0:10.0.0.23<0>
```

it's most likely not what you want if you have faster network interfaces available. But, of course, in some situations the Ethernet NIC is all you have, in which case the above is just fine - it'll be just very slow.

Sometimes, if the wrong interface ends up being used, the application might just hang.

If you have all the correct interfaces, plus some incorrect interfaces NCCL might work but at the slower speed.

If it's a cloud environment, typically your cloud provider should give you instructions on how to set things up correctly. If

they didn't then you need to at least ask them which network interfaces you need to use to setup NCCL.

While NCCL tries hard to auto-discover which interfaces it should use, if it fails to do so correctly you can then help it by telling it which interfaces to use or not to use:

- `NCCL_SOCKET_IFNAME` can be used to specify which `ifconfig` interfaces to include or exclude when not using InfiniBand. Here are some examples:

```
export NCCL_SOCKET_IFNAME=eth         # all interfaces starting with eth
export NCCL_SOCKET_IFNAME==eth0       # only eth0
export NCCL_SOCKET_IFNAME==eth0,eth1  # only interfaces eth0 and eth1
export NCCL_SOCKET_IFNAME=^docker     # do not use any interface starting with docker
export NCCL_SOCKET_IFNAME=^=docker0   # do not use interface docker0
```

The full doc is here.

- When using IB RDMA (IB Verbs interfaces), instead of `NCCL_SOCKET_IFNAME` use `NCCL_IB_HCA` env var which selects the interfaces for the collective communications. Examples:

```
export NCCL_IB_HCA=mlx5               # all ports of cards starting with mlx5
export NCCL_IB_HCA==mlx5_0:1,mlx5_1:1  # port 1 of mlx5_0 and mlx5_1
export NCCL_IB_HCA=^=mlx5_1,mlx5_4     # do not use mlx5_1 and mlx5_4
```

The full doc is here.

For example, often with IB, there will be additional interfaces like `mlx5_bond_0` which you don't want to be included in the NCCL comms. For example, this report would indicate that the wrong `[8]mlx5_bond_0:1/RoCE` interface was included and this would almost certainly lead to a low bandwidth:

```
NCCL INFO NET/IB : Using [0]mlx5_0:1/IB [1]mlx5_1:1/IB [2]mlx5_2:1/IB [3]mlx5_3:1/IB [4]mlx5_4:1/
IB [5]mlx5_5:1/IB [6]mlx5_6:1/IB [7]mlx5_7:1/IB [8]mlx5_bond_0:1/RoCE [RO]; OOB
ibp25s0:10.0.12.82<0>
```

There you'd exclude it with:

```
export NCCL_IB_HCA=^mlx5_bond_0:1
```

or alternatively you could list explicitly the interfaces you want, e.g.:

```
export NCCL_IB_HCA==mlx5_0,mlx5_1,mlx5_2,mlx5_3,mlx5_4,mlx5_5,mlx5_6,mlx5_7
```

As mentioned earlier, `ibstat` on one of the nodes will list the available interfaces if the fabric is InfiniBand; on any other RDMA fabric use `ibv_devices` or `ibv_devinfo`.

Since NCCL tries to automatically choose the best network interfaces, you only need to do the above if NCCL doesn't work or it's slow. In normal circumstances NCCL should work out of the box, without the user needing to do anything special.

Additionally, depending on which cloud is used, it's very likely that the provider may give you a slew of environment variables to set. If you set some of them incorrectly, NCCL might work slowly or not work at all.

Another typical problem users run into is when they try to reuse their NCCL setup they had working on cloud A on a cloud B. Often things don't translate and one has to carefully remove any previously set environment variables and set them correctly anew for the new cloud. This issue is likely to occur even if you're using the same cloud, but different types of instances, as some network setups are very specific for a given instance and won't work elsewhere.

Once you think you have set up the NCCL correctly, the next thing is to benchmark your connectivity and ensure that it matches the advertised speed (well, ~80% of it). Proceed to the benchmark chapter.

NCCL with docker containers

- Give enough resources by adding to the docker `run` these additional args: `--shm-size=1g --ulimit memlock=-1` (more details)
- Privileged access: sometimes you need to add `--privileged` to the docker `run` args.
- Having the docker image include the right packages, e.g. if using IB you'd want at least to install `libibverbs1 librdmacm1`

How to check if P2P is supported

Sometimes you need to know if the GPUs on your compute node support P2P access (Peer2Peer). Disabling P2P will typically lead to a slow intra-node connectivity.

You can see that on this particular 8x NVIDIA H100 node the P2P is supported:

```
$ nvidia-smi topo -p2p r
        GPU0    GPU1    GPU2    GPU3    GPU4    GPU5    GPU6    GPU7
 GPU0   X       OK      OK      OK      OK      OK      OK      OK
 GPU1   OK      X       OK      OK      OK      OK      OK      OK
 GPU2   OK      OK      X       OK      OK      OK      OK      OK
 GPU3   OK      OK      OK      X       OK      OK      OK      OK
 GPU4   OK      OK      OK      OK      X       OK      OK      OK
 GPU5   OK      OK      OK      OK      OK      X       OK      OK
 GPU6   OK      OK      OK      OK      OK      OK      X       OK
 GPU7   OK      OK      OK      OK      OK      OK      OK      X

Legend:

  X    = Self
  OK   = Status Ok
  CNS  = Chipset not supported
  GNS  = GPU not supported
  TNS  = Topology not supported
  NS   = Not supported
  U    = Unknown
```

On the other hand with this particular 2x NVIDIA L4 the P2P is not supported:

```
$ nvidia-smi topo -p2p r
```

```
        GPU0    GPU1
GPU0    X       CNS
GPU1    CNS     X
```

As you can see from the Legend,`CNS` signifies that "Chipset is not supported".

If you're using a high-end datacenter GPUs this is very unlikely to happen. Though some low-end datacenter GPUs may not support P2P like the example of L4 above.

For consumer-level GPUs there could be a variety of reasons for your GPU not being supported, often it's the IOMMU and/or ACS features being enabled. At other times it's just the driver version. And if you spend some time searching you might find someone hacking drivers to enable P2P in GPUs that shouldn't support P2P, like this 4090 P2P support repo.

To check if PCI Access Control Services (ACS) are enabled and to disable those follow this guide.

IOMMU can be disabled in the BIOS.

You can also check P2P support between specific GPUs using torch - here are we checking for GPUs 0 and 1:

```
python -c "import torch; print(torch.cuda.can_device_access_peer(torch.device('cuda:0'),
torch.device('cuda:1')))"
```

If there is no P2P support, the above would print `False`.

How to count NCCL calls

Enable NCCL debug logging for subsystems - collectives:

```
export NCCL_DEBUG=INFO
export NCCL_DEBUG_SUBSYS=COLL
```

if you're working in a slurm environment with many nodes you probably want to perform this only on rank 0, like so:

```
if [[ $SLURM_PROCID == "0" ]]; then
  export NCCL_DEBUG=INFO
  export NCCL_DEBUG_SUBSYS=COLL
fi
```

Assuming your logs were all sent to `main_log.txt`, you can then count how many of each collective call were performed with:

```
grep -a "NCCL INFO Broadcast" main_log.txt     | wc -l
2590
grep -a "NCCL INFO AllReduce" main_log.txt     | wc -l
5207
grep -a "NCCL INFO AllGather" main_log.txt     | wc -l
1849749
```

```
grep -a "NCCL INFO ReduceScatter" main_log.txt | wc -l
82850
```

It might be a good idea to first isolate a specific stage of the training, as loading and saving will have a very different pattern from training iterations.

So I typically first slice out one iteration. e.g. if each iteration log starts with: `iteration: ...` then I'd first do:

```
csplit main_log.txt '/iteration: /' "{*}"
```

and then analyze one of the resulting files that correspond to the iterations. By default it will be named something like `xx02`.

Useful NCCL Debug Environment Variables

The following env vars are most useful during debugging NCCL-related issues such as hanging and crashing. The full list of those can be found here.

`NCCL_DEBUG`

This is the most commonly used env var to debug networking issues.

Values:

- `VERSION` - Prints the NCCL version at the start of the program.
- `WARN` - Prints an explicit error message whenever any NCCL call errors out.
- `INFO` - Prints debug information
- `TRACE` - Prints replayable trace information on every call.

For example:

```
NCCL_DEBUG=INFO torchrun --nproc_per_node 2 --nnodes 1 torch-distributed-gpu-test.py
```

This will dump a lot of NCCL-related debug information, which you can then search online if you find that some problems are reported.

And `NCCL_DEBUG_FILE` should be very useful when using `NCCL_DEBUG` as the information is copious especially if using many nodes.

`NCCL_DEBUG_FILE`

When using `NCCL_DEBUG` env var, redirect all NCCL debug logging output to a file.

The default is `stdout`. When using many GPUs it can be very useful to save each process' debug info into its own log file, which can be done like so:

```
NCCL_DEBUG_FILE=/path/to/nccl-log.%h.%p.txt
```

- `%h` is replaced with the hostname
- `%p` is replaced with the process PID.

If you then need to analyze hundreds of these at once, here are some useful shortcuts:

- grep for a specific match and also print the file and line number where it was found:

```
grep -n "Init COMPLETE" nccl-log*
```

- show `tail -1` of all nccl log files followed by the name of each file

```
find . -name "nccl*" -exec sh -c 'echo "$(tail -1 "$1") ($1)"' _ {} \;
```

NCCL_DEBUG_SUBSYS

`NCCL_DEBUG_SUBSYS` used in combination with `NCCL_DEBUG` tells the latter which subsystems to show. Normally you don't have to specify this variable, but sometimes the developers helping you may ask to limit the output to only some sub-systems, for example:

```
NCCL_DEBUG_SUBSYS=INIT,GRAPH,ENV,TUNING
```

NCCL_IB_HCA

When the collectives run over IB Verbs rather than sockets, this is the variable that selects which adapters they use - the RDMA counterpart to `NCCL_SOCKET_IFNAME`. Run `ibv_devinfo` to see what the node has.

There are three forms:

```
export NCCL_IB_HCA=mlx5               # every port of every card whose name starts with mlx5
export NCCL_IB_HCA==mlx5_0:1,mlx5_1:1  # exactly port 1 of mlx5_0 and of mlx5_1
export NCCL_IB_HCA=^=mlx5_1,mlx5_4     # everything except mlx5_1 and mlx5_4
```

A worked example of using the exclusion form to route around a single problem adapter is in How to diagnose NCCL multi-gpu and multi-node connectivity issues. The full doc is here.

NCCL_P2P_DISABLE

Disables P2P comms - e.g. NVLink won't be used if there is one and the performance will be much slower as a result of that. Normally you don't want this but in a pinch sometimes this can be useful during debug.

NCCL_SOCKET_IFNAME

This one is very useful if you have multiple network interfaces and you want to choose a specific one to be used.

`NCCL_SOCKET_IFNAME` selects which sockets NCCL may use for bootstrap and for collectives that run over TCP/IP - it does not pick InfiniBand Verbs adapters (those are `NCCL_IB_HCA`).

For example, to restrict NCCL to Ethernet interfaces:

```
NCCL_SOCKET_IFNAME=eth
```

This env var can be used at times to debug connectivity issues, if say one of the interfaces is firewalled, and perhaps the others aren't and can be tried instead. Or if you are not sure whether some problem is related to the network interface or something else, so it helps to test other interfaces to invalidate that the issue comes from network.

Networking Benchmarks

Tools

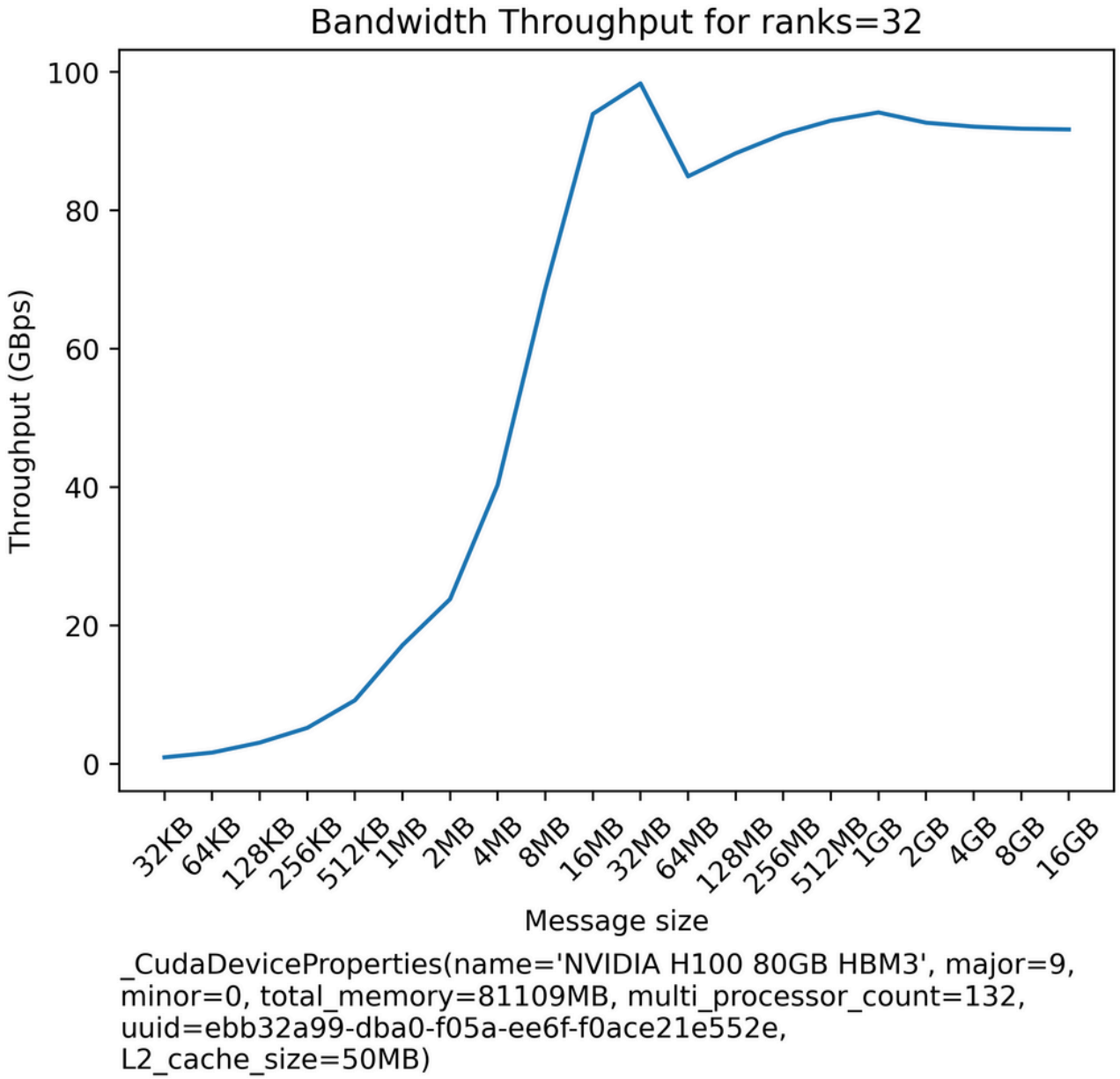
all_reduce benchmark

all_reduce_bench.py - a tool to benchmark the real network bandwidth while performing `all_reduce` on a largish amount of data. This is useful for finding out what one gets in reality as compared to the advertised spec. Somewhat similar to `nccl-tests`, but requires just PyTorch to run.

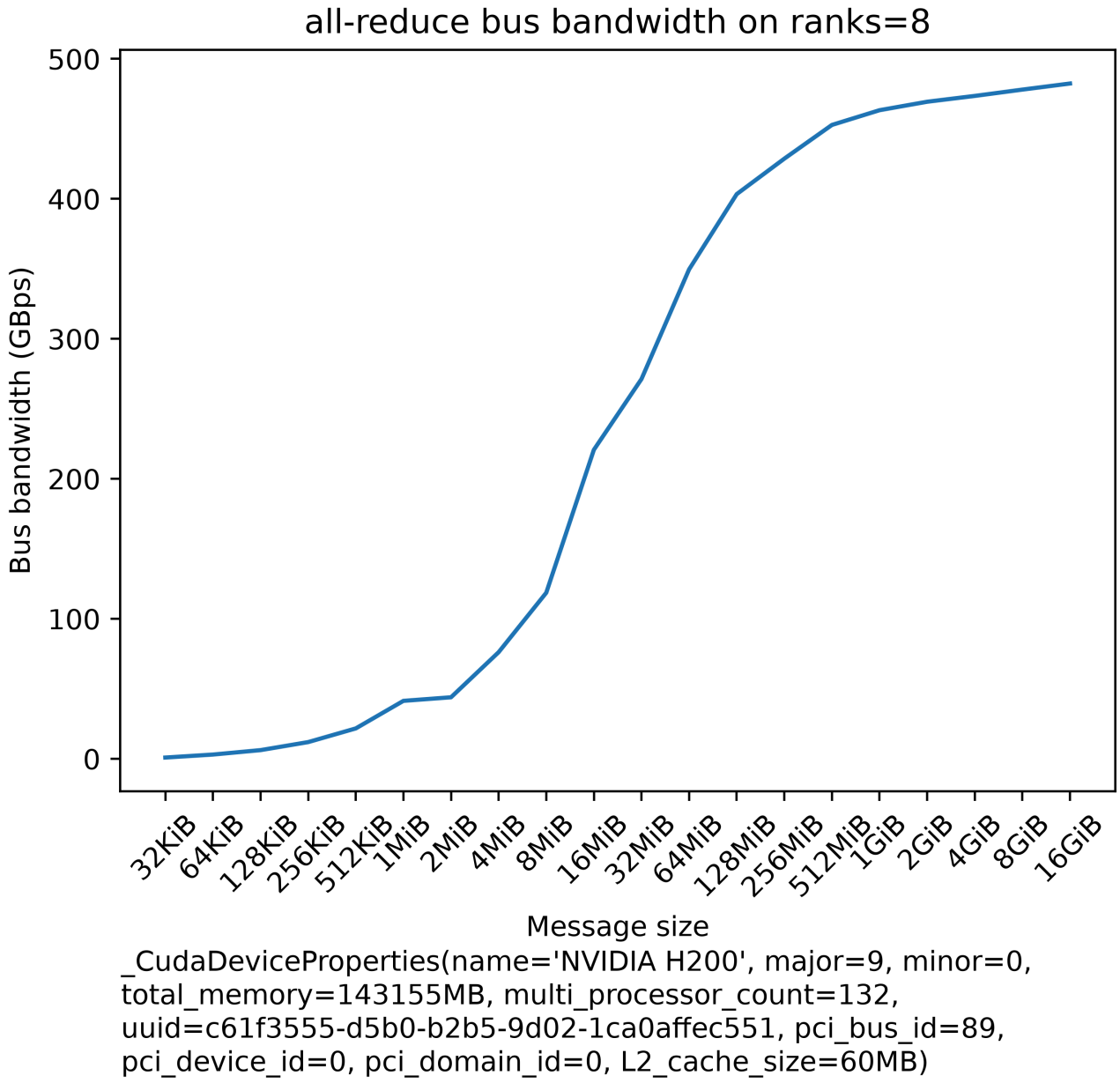
It generates output like this:

```
| payload |    busbw   |    algbw   |
| ------: | ---------: | ---------: |
|   32KiB |   0.92GBps |   0.48GBps |
|   64KiB |   1.61GBps |   0.83GBps |
|  128KiB |   3.05GBps |   1.58GBps |
|  256KiB |   5.18GBps |   2.67GBps |
|  512KiB |   9.17GBps |   4.73GBps |
|    1MiB |  17.13GBps |   8.84GBps |
|    2MiB |  23.79GBps |  12.28GBps |
|    4MiB |  40.30GBps |  20.80GBps |
|    8MiB |  68.62GBps |  35.42GBps |
|   16MiB |  93.93GBps |  48.48GBps |
|   32MiB |  98.34GBps |  50.76GBps |
|   64MiB |  84.90GBps |  43.82GBps |
|  128MiB |  88.23GBps |  45.54GBps |
|  256MiB |  91.01GBps |  46.97GBps |
|  512MiB |  92.95GBps |  47.98GBps |
|    1GiB |  94.15GBps |  48.59GBps |
|    2GiB |  92.66GBps |  47.83GBps |
|    4GiB |  92.09GBps |  47.53GBps |
|    8GiB |  91.80GBps |  47.38GBps |
|   16GiB |  91.69GBps |  47.32GBps |
```

And it also creates a plot:



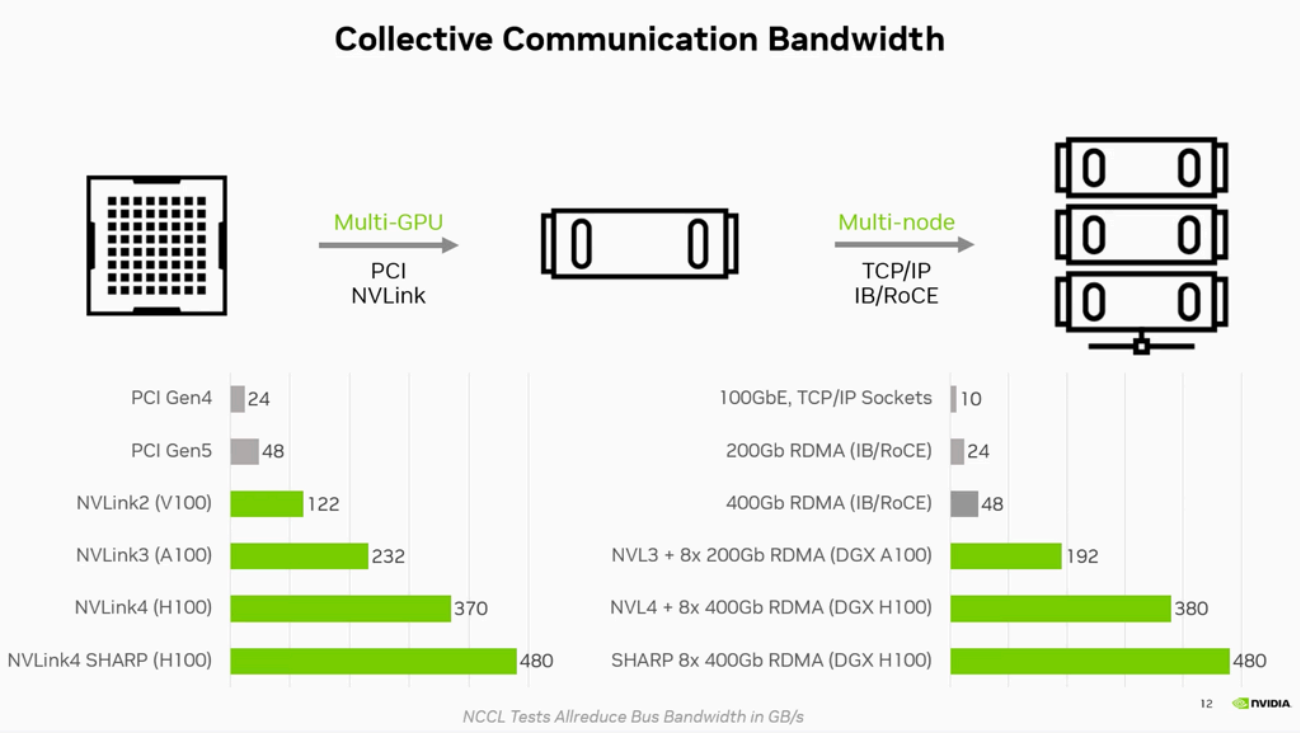
Here is the same benchmark on a single 8x H200 node (`torch=2.9.1+cu130`, `cuda=13.0`, `nccl=2.27.7`, 5 warmup and 20 trial iterations per payload, 47 seconds for the whole sweep):



Note the linear y-axis compresses everything below ~100GBps into the bottom of the plot, so the small-payload end - the part that matters for gradient bucketing - is easier to read off the printed table than off the curve. That sweep tops out at 482.26GBps, which is *above* the 450GBps unidirectional NVLink 4 spec rather than below it; see SHARP for why, and for what the same node measures with it disabled.

For launching examples and notes please see the top of all_reduce_bench.py.

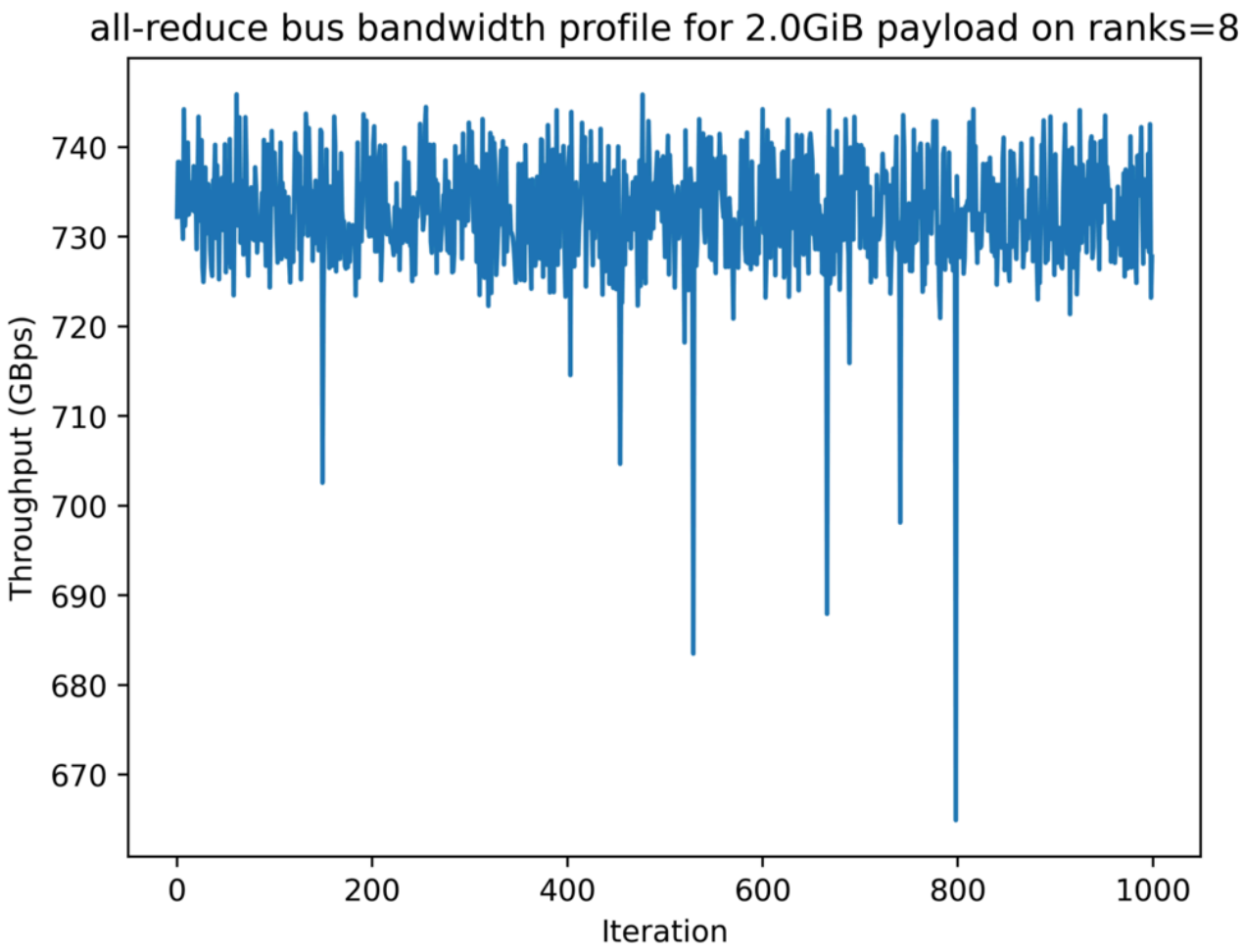
This table should give a good sense for what scores you should expect for all-reduce collective on a well-tuned network (left is intra-node and right is inter-node):



source

If you're benchmarking a different collective the expected bandwidth can be very different from the above all-reduce results. This presentation also gives point-to-point communication bandwidth expectations.

To check the stability of all-reduce over time, rather than averaging the results, you can profile a single payload size with these 2 flags `--profile_stability --payload_size_in_gib 0.5` (change the last value to the desired payload size in GiB). Beware that a typical ML workload doesn't call all-reduce back to back non-stop so this approach puts the network through a stress test, which is a somewhat non-typical workload. But it can still show if the network has issues with sustained load. Here is an example of a plot generated on a 8x B200 with a payload of 2GiB:




```
_CudaDeviceProperties(name='NVIDIA B200', major=10, minor=0,
total_memory=182631MB, multi_processor_count=148,
uuid=c82c8bf2-badb-75b7-7435-c5eef7a58a90, pci_bus_id=81,
pci_device_id=0, pci_domain_id=0, L2_cache_size=126MB)
```

all_gather_object vs all_reduce

all_gather_object_vs_all_reduce.py - a quick benchmark showing 23x speed up when moving from `all_gather_object` to `all_reduce` when collecting completion status from the process group. e.g. when implementing some sort of all-processes-are-done flag. This technique is usually used for synchronizing gpus when they may complete at different number of iterations - which one needs for inference over multiple DP channels, or when one wants to sync a `StopIteration` event in `DataLoader`. See also all_gather_object_vs_all_gather.py.

all_reduce latency comparison

all_reduce_latency_comp.py - exemplifies how 1x 4GB reduction is much faster than 1000x 4MB reductions.

nccl-tests

NVIDIA/nccl-tests benchmarks collectives - `all-reduce`, `all-gather`, `reduce-scatter` and the rest. It reports the same `busbw`/`algbw` columns as all_reduce_bench.py and the two agree closely, but it covers every collective rather than just

`all-reduce`.

`MPI=0` is fine for a single node, and `NCCL_HOME` points at whichever NCCL you want to test - the one bundled with PyTorch being the convenient choice, since that is what your training will actually use:

```
git clone https://github.com/NVIDIA/nccl-tests
cd nccl-tests
make -j MPI=0 NCCL_HOME=$(python -c "import torch, os; print(os.path.dirname(torch.__file__) +
'/lib')")
```

That puts one binary per collective under `build/` - `all_reduce_perf`, `all_gather_perf`, `reduce_scatter_perf`, `alltoall_perf` and others. If they fail to find `libnccl` at run time, add the same directory to `LD_LIBRARY_PATH`. Add `-z 1` for a blocking run, which matches how `all_reduce_bench.py` measures.

nvbandwidth

NVIDIA/nvbandwidth measures point-to-point bandwidth between hosts and accelerators - the closest thing to a direct reading of a single link, as opposed to a collective's aggregate:

```
git clone https://github.com/NVIDIA/nvbandwidth
cd nvbandwidth
cmake . && make
```

Run it with no arguments for the full sweep, `./nvbandwidth -l` to list the testcases, or `-t <testcase>` to run just one. `-i N` raises the iteration count from its default of 3.

note: `host_to_device_memcpy_ce` measures whatever the host-to-device path happens to be on that platform - PCIe on an x86 host with PCIe-attached accelerators, NVLink-C2C on a Grace-Blackwell system. Same command, an order of magnitude apart, so read the number against the fabric the machine actually uses.

p2pBandwidthLatencyTest

p2pBandwidthLatencyTest from CUDA samples is a low-level accelerator-to-accelerator benchmark:

```
git clone https://github.com/NVIDIA/cuda-samples/
cd cuda-samples/cpp/5_Domain_Specific/p2pBandwidthLatencyTest
nvcc -o p2pBandwidthLatencyTest p2pBandwidthLatencyTest.cu -I ../../../Common
```

note: this repository reorganized its layout - the samples used to live under `Samples/` and are now under `cpp/`. If the `cd` fails, `find . -name p2pBandwidthLatencyTest.cu` will locate it. `Common` is still at the repository root, so the `-I` path is unchanged.

Crucial reproducibility requirements

The most important requirements for a series of successful experiments is to be able to reproduce the experiment environment again and again while changing only one or a few setup variables.

Therefore when you try to figure out whether some change will improve performance or make it worse, you must figure out how to keep things stable.

For example, you need to find a way to prevent the network usage from fluctuations. When we were doing performance optimizations for 108B pre-BLOOM experiments it was close to impossible to perform, since we were on a shared internode network and the exact same setup would yield different throughput depending on how many other users used the network. It was not working. During BLOOM-176B we were given a dedicated SLURM partition with an isolated network where the only traffic was ours. Doing the performance optimization in such environment was just perfect.

Network throughput

It's critical to understand your particular model size and framework requirements with regard to network bandwidth, throughput and latency. If you underpay for network you will end up having idle gpus and thus you wasted money and time. If you overpay for very fast network, but your gpus are slow, then again you wasted money and time.

If your network is very slow, your training is likely to be network-bound and many improvements in the training setup will not help with the improving performance.

Note: The EAI cookbook contains a set of communication benchmarks for each collective that you can use to quickly measure the throughput of your internode or intranode network.

Here is a simple all-reduce benchmark that you can use to quickly measure the throughput of your internode network:

all_reduce_bench.py

On CSPs that have enabled SLURM Pyxis Container Plugin, such as CoreWeave, Crusoe, AWS, Oracle, Azure, GCP, etc, `all_reduce_bench.py` can be easily ran & reproduced via the following command:

```
sbatch -n <num_of_nodes> ./all_reduce_bench_pyxis.sbatch
```

Usually benchmarking at least 4 nodes is recommended, but, of course, if you already have access to all the nodes you will be using during the training, benchmark using all of the nodes.

If you do not have access to a pyxis SLURM environment, to run it on 4 nodes:

```
GPUS_PER_NODE=8
NNODES=4
MASTER_ADDR=$(scontrol show hostnames $SLURM_JOB_NODELIST | head -n 1)
MASTER_PORT=6000
python -u -m torch.distributed.run \
    --nproc_per_node $GPUS_PER_NODE \
    --nnodes $NNODES \
    --rdzv_endpoint $MASTER_ADDR:$MASTER_PORT \
    --rdzv_backend c10d \
    --max_restarts 0 \
    --role `hostname -s`: \
    --tee 3 \
    all_reduce_bench.py
```

Notes:

- adapt `MASTER_ADDR` to rank 0 hostname if it's not a SLURM environment where it's derived automatically.

Here is how to launch it in a SLURM env with 4 nodes:

```
salloc --partition=mypartition --nodes=4 --ntasks-per-node=1 --cpus-per-task=48 --gres=gpu:8
--time=1:00:00 bash
srun --cpus-per-task=$SLURM_CPUS_PER_TASK --gres=gpu:8 --nodes=4 --tasks-per-node=1 python -u -m
torch.distributed.run --nproc_per_node=8 --nnodes 4 --rdzv_endpoint $(scontrol show hostnames
$SLURM_JOB_NODELIST | head -n 1):6000 --rdzv_backend c10d all_reduce_bench.py
```

Notes:

- You are likely to need to adapt `--cpus-per-task` and `--partition` arguments there.
- You do `salloc` once and then can repeat `srun` multiple times on the same allocation.

You may get results anywhere between 5Gbps and 6800Gbps (as of 2026-08), and the payload size matters as much as the hardware does - `busbw` climbs by orders of magnitude from a small payload to a large one on the very same setup. In the measured tables under Inter-node speed depends on intra-node speed, at a 16GiB payload a single B200 node reaches 845.67GBps and four nodes 381.80GBps - about 6800Gbps and 3050Gbps - while at 32KiB those same runs report 1.20GBps and 0.01GBps. So always compare like payload with like. The minimal speed to prevent being network bound will depend on your particular training framework, but typically you'd want at least 400Gbps or higher. Though we trained BLOOM on 50Gbps.

Frameworks that shard weights and optim stages like DeepSpeed w/ ZeRO Stage-3 do a lot more traffic than frameworks like Megatron-DeepSpeed which do tensor and pipeline parallelism in addition to data parallelism. The latter ones only send activations across and thus don't need as much bandwidth. But they are much more complicated to set up and run.

Of course, an efficient framework will overlap communications and compute, so that while one stage is fetching data, the other stage in parallel runs computations. So as long as the communication overhead is smaller than compute the network requirements are satisfied and don't have to be super fantastic.

To get reasonable GPU throughput when training at scale (64+GPUs) with DeepSpeed ZeRO Stage 3 with V100s

1. 100Gbps is not enough
2. 200-400Gbps is ok
3. 800-1000Gbps is ideal

full details

Of course, the requirements are higher for A100 gpu nodes and even higher for H100s (but no such benchmark information has been shared yet).

Extrapolating benchmark results from several nodes to many

As it's often not easy to benchmark hundreds of nodes, often we try to benchmark interconnect performance using, say, 4 nodes. I wasn't sure whether this would give the correct indication for when 40 or 400 nodes will be used so I asked about it here and the answer was:

> Extrapolating at scale is not that hard for ring and tree (we have a function in `tuning.cc` predicting it, based on the ring linear latency and the tree log latency with reduced BW). Now as you scale, there are many factors which may cause your real performance to be very far off the prediction, like routing. Also note on an IB network you'll be able to use SHARP; that way your latency stays mostly constant as you scale, your bandwidth doesn't degrade much either, and you're always better than both ring and tree.

Disable Access Control Services

PCI Access Control Services (ACS) used for IO virtualization (also known as VT-d or IOMMU) force P2P PCIe transactions to

go up through the PCIe Root Complex, which does not enable GDS to bypass the CPU on paths between a network adapter or NVMe and the GPU in systems that include a PCIe switch.

For the optimal GDS performance, disable ACS by following these instructions here. Here are some additional notes

Please note that if you're using Virtual machines you can't disable ACS as it's a required feature. To run with maximum performance inside virtual machines, Address Translation Service (ATS) needs to be enabled in network adapters.

Performance-Oriented NCCL Environment Variables

While NCCL is excellent at automatically figuring out the best performance for any given network, sometimes it needs some help, in which case the following NCCL env vars are used to tune up performance. Let's look at a few common ones you might want to be aware of, and the full list of those can be found here.

Note that some `NCCL_IB_*` env vars apply to RoCEv2 networks as well.

`NCCL_ALGO`

This one defines which algorithms NCCL will use. Typically it's one of:

1. Tree
2. Ring
3. CollnetDirect and CollnetChain (IB SHARP)
4. NVLS (NVLink SHARP)

I was asking questions about how a user can do the optimization and was told at this NCCL Issue that basically the user shouldn't try to optimize anything as NCCL has a ton of smart algorithms inside that will try to automatically switch from one algorithm to another depending on a concrete situation.

Sylvain Jeaugey shared:

> There used to be a static threshold, but it's been replaced by a more complex tuning system. The new system builds a model of the latency and bandwidth of each algorithm/protocol combination (that's many, many combinations) and decides which one should perform best depending on the size. So there is no longer an env var and a static value, which is good because the performance of each algorithm depends on the number of nodes and number of GPUs per node and therefore we need to navigate a 2D space of algo/protocols which isn't easy. You can always force one algorithm with `NCCL_ALGO=TREE` and `NCCL_ALGO=RING` and see what performance you get and whether NCCL switches at the right point. I know it's hard to understand, but it's also the best solution we found to have the best performance across all platforms and users without users having to manually tune the switch points. Downside is, if you want to manually tune things, you can't.

If you use `NCCL_ALGO` you need to list the algorithms to consider, but otherwise you have no control over it. So, really, this is only useful if you want to make sure that one of the algorithms isn't used.

When asking about which algorithm is better, I received:

> Roughly speaking, ring is superior in terms of peak bandwidth (except on 2 nodes), tree is superior in terms of base latency (especially as we scale). `Bandwidth = Size / Time`, so whether you look at the time or the bandwidth for a given size, it will be a combination of both the peak bandwidth and the base latency. For a fixed size, as you scale, the base latency of ring will become prevalent and tree will be better.

There is also an algo named `NVLS`, which uses NVLink SHARP to do the reduction inside the switch and can therefore report

more than the wire spec - with NVLink 4.0 (450GBps) an `all-reduce` benchmark clocks 480GBps. `NVLSTree` (NCCL 2.18+) is the inter-node counterpart and requires IB or RoCE. See SHARP for when it engages, what it is worth, and why `busbw` stops describing the wire once it does.

And if you would like to know which algo is being used, `NCCL_DEBUG=INFO` combined with `NCCL_DEBUG_SUBSYS=INIT,TUNING` reports the selection per payload size:

```
NCCL_DEBUG=INFO NCCL_DEBUG_SUBSYS=INIT,TUNING NCCL_DEBUG_FILE=/tmp/nccl.%h.%p.log \
./build/all_reduce_perf -b 32k -e 16G -f 2 -g 8
grep -ihoE "AllReduce: [0-9]+ Bytes -> Algo [A-Z]+ proto [A-Z0-9]+" /tmp/nccl.*.log | sort -u
```

On an 8x H200 node that prints lines like `AllReduce: 2097152 Bytes -> Algo NVLS proto SIMPLE`, and reveals where NCCL switches over - `RING` with the `LL` protocol for payloads up to 1MiB, then `NVLS` with `SIMPLE` from 2MiB up. `NCCL_DEBUG_FILE` keeps all of this out of the benchmark's own output, which otherwise gets buried.

Setting `NCCL_ALGO` explicitly is still worth doing, but for a different purpose - measuring what each algorithm delivers on your hardware, rather than discovering which one NCCL chose.

NCCL_CROSS_NIC

The `NCCL_CROSS_NIC` variable controls whether NCCL should allow rings/trees to use different NICs, causing inter-node communication to use different NICs on different nodes.

To maximize inter-node communication performance when using multiple NICs, NCCL tries to communicate between same NICs between nodes, to allow for network design where each NIC from each node connects to a different network switch (network rail), and avoid any risk of traffic flow interference. The NCCL_CROSS_NIC setting is therefore dependent on the network topology, and in particular depending on whether the network fabric is rail-optimized or not.

This has no effect on systems with only one NIC.

Values accepted:

- 0: Always use the same NIC for the same ring/tree, to avoid crossing network rails. Suited for networks with per NIC switches (rails), with a slow inter-rail connection. Note there are corner cases for which NCCL may still cause cross-rail communication, so rails still need to be connected at the top.
- 1: Do not attempt to use the same NIC for the same ring/tree. This is suited for networks where all NICs from a node are connected to the same switch, hence trying to communicate across the same NICs does not help avoiding flow collisions.
- 2: (Default) Try to use the same NIC for the same ring/tree, but still allow for it if it would result in better performance.

NCCL_IB_QPS_PER_CONNECTION

This is relevant if you're on a multi-layer InfiniBand or RoCEv2 network.

`NCCL_IB_QPS_PER_CONNECTION` defines the number of IB queue pairs to use for each connection between two ranks. This can be useful on multi-level fabrics which need multiple queue pairs to have good routing entropy. In other words, when your jobs are crossing spine or super-spine switches.

By default it is set to `1`, but having a higher number might benefit throughput.

Depends on the size of the network. you could start with something like `4` for any cluster over 64 GPUs (i.e. any cluster that's bigger than the radix (number of ports) of its IB switch (e.g. the IB NDR switch radix is 64.)

Ideally you'd ask your cloud provider if they have already researched the best value, but if they didn't you can do it yourself, albeit it might be use-case specific.

The other gotcha is that when the value is higher than `1` an additional GPU memory will be consumed.

`NCCL_MIN_CTAS` and `NCCL_MAX_CTAS`

Cooperative Thread Array (CTA) implements CUDA thread blocks - You can read about it here.

In the past these 2 env vars were called `NCCL_MIN_NCHANNELS` and `NCCL_MAX_NCHANNELS`.

Because in the CUDA world compute and communication operations share the same limited number of SMs per GPU, if too many SMs are used for compute, the comms will be blocked and vice versa. Since ideally compute and comms should overlap and not block each other finding the right balance is important.

The CTA value is derived algorithmically by NCCL, but the default behavior can be overridden by setting the lower and upper limits via the env vars: `NCCL_MIN_CTAS` and `NCCL_MAX_CTAS`. And then NCCL's tuner will be limited to choose the best value in the user-imposed range. The same can be accomplished from the program using `pg_options` in `torch.distributed.init_process_group` via `ncclConfig_t`'s `minCTAs` and `maxCTAs` (other process group creation functions have `pg_options` as well). The latter approach allows you to set different CTA settings to different process groups, whereas the env vars will apply globally to all process groups.

Here is an example that directly sets both values to `32` per process group:

```
import torch
nccl_options = torch.distributed.ProcessGroupNCCL.Options()
nccl_options.config.min_ctas = 32
nccl_options.config.max_ctas = 32
torch.distributed.init_process_group(..., pg_options=nccl_options)
```

In order to find the best performance to experiment with different values against a specific benchmark of choice, that emulates the intended workload, you could set both config options to the same value and then bisect on a range of 1 to 64 or similar.

InfiniBand

InfiniBand adaptive routing

Make sure your cloud provider enables IB adaptive routing which could greatly improve the performance.

For nuances see this paper: Adaptive Routing in InfiniBand Hardware.

Orchestration

There are many container/accelerator orchestration solutions - many of which are open source.

So far I have been working with SLURM and Kubernetes.

SLURM

- SLURM - Simple Linux Utility for Resource Management, which you're guaranteed to find on most HPC environments and typically it's supported by most cloud providers. It has been around for more than 2 decades

SLURM on Kubernetes:

These are frameworks allowing you to run SLURM on top of Kubernetes:

- Slinky - a framework from the creators of SLURM.
- Soperator - from Nebius
- CoreWeave never released their SUNK open source, so it's only available to its customers.

Kubernetes

The other most popular orchestrator is Kubernetes:

- Kubernetes - also known as K8s, is an open source system for automating deployment, scaling, and management of containerized applications. Here is a good comparison between SLURM and K8s.

Other orchestration frameworks

Here are various other less popular than SLURM and K8s, but still very mighty orchestration solutions:

- dstack is a lightweight, open-source alternative to Kubernetes & SLURM, simplifying AI container orchestration with multi-cloud & on-prem support. It natively supports NVIDIA, AMD, & TPU.
- SkyPilot is a framework for running AI and batch workloads on any infra, offering unified execution, high cost savings, and high GPU availability.
- OpenHPC provides a variety of common, pre-built ingredients required to deploy and manage an HPC Linux cluster including provisioning tools, resource management, I/O clients, runtimes, development tools, containers, and a variety of scientific libraries.
- run.ai - got acquired by NVIDIA and is planned to be open sourced soon.
- Docker Swarm is a container orchestration tool.
- IBM Platform Load Sharing Facility (LSF) Suites is a workload management platform and job scheduler for distributed high performance computing (HPC).

Working in SLURM Environment

Unless you're lucky and you have a dedicated cluster that is completely under your control chances are that you will have to use SLURM to timeshare the GPUs with others. But, often, if you train at HPC, and you're given a dedicated partition you still will have to use SLURM.

The SLURM abbreviation stands for: **Simple Linux Utility for Resource Management** - though now it's called The SLURM Workload Manager. It is a free and open-source job scheduler for Linux and Unix-like kernels, used by many of the world's supercomputers and computer clusters.

These chapters will not try to exhaustively teach you SLURM as there are many manuals out there, but will cover some specific nuances that are useful to help in the training process.

- SLURM For Users - everything you need to know to do your training in the SLURM environment.
- SLURM Administration - if you're unlucky to need to also manage the SLURM cluster besides using it, there is a growing list of recipes in this document to get things done faster for you.
- Performance - SLURM performance nuances.
- Launcher scripts - how to launch with `torchrun`, `accelerate`, pytorch-lightning, etc. in the SLURM environment

SLURM Administration

Run a command on multiple nodes

1. to avoid being prompted with:

```
Are you sure you want to continue connecting (yes/no/[fingerprint])?
```

for every new node you haven't logged into yet, you can disable this check with:

```
echo "Host *" >> ~/.ssh/config
echo "  StrictHostKeyChecking no" >> ~/.ssh/config
```

Of course, check if that's secure enough for your needs. I'm making an assumption that you're already on the SLURM cluster and you're not ssh'ing outside of your cluster. You can choose not to set this and then you will have to manually approve each new node.

2. Install `pdsh`

You can now run the wanted command on multiple nodes.

For example, let's run `date`:

```
$ PDSH_RCMD_TYPE=ssh pdsh -w node-[21,23-26] date
node-25: Sat Oct 14 02:10:01 UTC 2023
node-21: Sat Oct 14 02:10:02 UTC 2023
node-23: Sat Oct 14 02:10:02 UTC 2023
node-24: Sat Oct 14 02:10:02 UTC 2023
node-26: Sat Oct 14 02:10:02 UTC 2023
```

Let's do something more useful and complex. Let's kill all GPU-tied processes that didn't exit when the SLURM job was canceled:

First, this command will give us all process ids that tie up the GPUs:

```
nvidia-smi --query-compute-apps=pid --format=csv,noheader | sort | uniq
```

So we can now kill all those processes in one swoop:

```
 PDSH_RCMD_TYPE=ssh pdsh -w node-[21,23-26]  "nvidia-smi --query-compute-apps=pid
--format=csv,noheader | sort | uniq | xargs -n1 sudo kill -9"
```

SLURM settings

Show the slurm settings:

```
sudo scontrol show config
```

The config file is `/etc/slurm/slurm.conf` on the slurm controller node.

Once `slurm.conf` was updated to reload the config run:

```
sudo scontrol reconfigure
```

from the controller node.

Node management

Changing the state of the node

The change is performed by `scontrol update`

Examples:

To undrain a node that is ready to be used:

```
scontrol update nodename=node-5 state=idle
```

To remove a node from the SLURM's pool:

```
scontrol update nodename=node-5 state=drain reason="maintenance"
```

Undrain nodes killed due to slow process exit

Sometimes processes are slow to exit when a job has been canceled. If the SLURM was configured not to wait forever it'll automatically drain such nodes. But there is no reason for those nodes to not be available to the users.

So here is how to automate it.

The key is to get the list of nodes that are drained due to `"Kill task failed"`, which is retrieved with:

```
sinfo -R | grep "Kill task failed"
```

now extract and expand the list of nodes, check that the nodes are indeed user-process free (or try to kill them first) and then undrain them.

Earlier you learned how to run a command on multiple nodes which we will use in this script.

Here is the script that does all that work for you: undrain-good-nodes.sh

Now you can just run this script and any nodes that are basically ready to serve but are currently drained will be switched to `idle` state and become available for the users to be used.

Auto-reboot

If the nodes need to be rebooted safely (e.g. if the image has been updated), adapt the list of the node and run:

```
scontrol reboot ASAP node-[1-64]
```

For each of the non-idle nodes this command will wait till the current job ends, then reboot the node and bring it back up to `idle`.

Note that you need to have:

```
RebootProgram = "/sbin/reboot"
```

set in `/etc/slurm/slurm.conf` on the controller node for this to work (and reconfigure the SLURM daemon if you have just added this entry to the config file).

Job management

Modify a job's timelimit

To set a new timelimit on a job, e.g., 2 days:

```
scontrol update JobID=$SLURM_JOB_ID TimeLimit=2-00:00:00
```

To add additional time to the previous setting, e.g. 10 more hours.

```
scontrol update JobID=$SLURM_JOB_ID TimeLimit=+10:00:00
```

When something goes wrong with SLURM

Analyze the events log in the SLURM's log file:

```
sudo cat /var/log/slurm/slurmctld.log
```

This, for example, can help to understand why a certain node got its jobs canceled before time or the node got removed completely.

SLURM for Users

Quick start

Simply copy this example.slurm and adapt it to your needs.

SLURM partitions

In this doc we will use an example setup with these 2 cluster names:

- `dev`
- `prod`

To find out the hostname of the nodes and their availability, use:

```
sinfo -p dev
sinfo -p prod
```

The path to `slurm.conf` is site-dependent (often under `/etc/slurm/` or `/opt/slurm/etc/`); to see where this cluster loads it from:

```
scontrol show config | grep -i slurmconf
```

To see the configuration of all partitions:

```
scontrol show partition
```

Wait time for resource granting

```
squeue -u `whoami` --start
```

will show when any pending jobs are scheduled to start.

They may start sooner if others cancel their reservations before the end of the reservation.

Request allocation via dependency

To schedule a new job when one more of the currently scheduled job ends (regardless of whether it still running or not started yet), use the dependency mechanism, by telling `sbatch` to start the new job once the currently running job succeeds, using:

```
sbatch --dependency=CURRENTLY_RUNNING_JOB_ID tr1-13B-round1.slurm
```

Using `--dependency` may lead to shorter wait times that using `--begin`, since if the time passed to `--begin` allows even for a few minutes of delay since the stopping of the last job, the scheduler may already start some other jobs even if their priority is lower than our job. That's because the scheduler ignores any jobs with `--begin` until the specified time arrives.

Make allocations at a scheduled time

To postpone making the allocation for a given time, use:

```
salloc --begin=YYYY-MM-DDTHH:MM:SS
```

Same for `sbatch`.

It will simply put the job into the queue at the requested time, as if you were to execute this command at this time. If resources are available at that time, the allocation will be given right away. Otherwise it'll be queued up.

Sometimes the relative begin time is useful. And other formats can be used. Examples:

```
--begin now+2hours
--begin=16:00
--begin=now+1hour
--begin=now+60  # seconds by default
--begin=2010-01-20T12:34:00
```

the time-units can be `seconds` (default), `minutes`, `hours`, `days`, or `weeks`.

Preallocated node without time 60min limit

This is very useful for running repetitive interactive experiments - so one doesn't need to wait for an allocation to progress. so the strategy is to allocate the resources once for an extended period of time and then running interactive `srun` jobs using this allocation.

set `--time` to the desired window (e.g. 6h):

```
salloc --partition=dev --nodes=1 --ntasks-per-node=1 --cpus-per-task=96 --gres=gpu:8
--time=6:00:00 bash
salloc: Pending job allocation 1732778
salloc: job 1732778 queued and waiting for resources
salloc: job 1732778 has been allocated resources
salloc: Granted job allocation 1732778
```

now use this reserved node to run a job multiple times, by passing the job id of `salloc`:

```
srun --jobid $SLURM_JOBID --pty bash
```

if run from inside `bash` started via `salloc`. But it can be started from another shell, but then explicitly set `--jobid`.

if this `srun` job timed out or manually exited, you can re-start it again in this same reserved node.

`srun` can, of course, call the real training command directly and not just `bash`.

Important: when allocating a single node, the allocated shell is not on the node (it never is). You have to find out the hostname of the node (reported when the allocation is granted, or via `squeue`) and then `ssh` to it.

When finished, to release the resources, either exit the shell started in `salloc` or `scancel JOBID`.

This reserved node will be counted towards hours usage the whole time it's allocated, so release as soon as done with it.

Actually, if this is just one node, then it's even easier to not use `salloc` but to use `srun` in the first place, which will both allocate and give you the shell to use:

```
srun --pty --partition=dev --nodes=1 --ntasks=1 --cpus-per-task=96 --gres=gpu:8 --time=60 bash
```

Hyper-Threads

By default, if the cpu has Hyper-Threads (HT) enabled, SLURM will use it. If you don't want to use HT you have to specify `--hint=nomultithread`.

footnote: HT is Intel-specific naming, the general concept is simultaneous multithreading (SMT)

For example for a cluster with 2 cpus per node with 24 cores and 2 hyper-threads each, there is a total of 96 hyper-threads or 48 cpu-cores available. Therefore to utilize the node fully you'd need to configure either:

```
#SBATCH --cpus-per-task=96
```

or if you don't want HT:

```
#SBATCH --cpus-per-task=48
#SBATCH --hint=nomultithread
```

This last approach will allocate one thread per core and in this mode there are only 48 cpu cores to use.

Note that depending on your application there can be quite a performance difference between these 2 modes. Therefore try both and see which one gives you a better outcome.

On some setups like AWS the all-reduce throughput degrades dramatically when `--hint=nomultithread` is used! Whereas on some other setups the opposite is true - the throughput is worse without HT!

To check if your instances has HT enabled, run:

```
$ lscpu | grep Thread
Thread(s) per core: 2
```

If it's 2 then it is HT-enabled, if it's 1 then it isn't.

Reuse allocation

e.g. when wanting to run various jobs on identical node allocation.

In one shell:

```
salloc --partition=prod --nodes=16 --ntasks=16 --cpus-per-task=96 --gres=gpu:8 --time=3:00:00
bash
echo $SLURM_JOBID
```

In another shell:

```
export SLURM_JOBID=<JOB ID FROM ABOVE>
srun --jobid=$SLURM_JOBID ...
```

You may need to set `--gres=gpu:0` to run some diagnostics job on the nodes. For example, let's check shared memory of all the hosts:

```
srun --jobid 631078 --gres=gpu:0 bash -c 'echo $(hostname) $(df -h | grep shm)'
```

Specific nodes selection

To exclude specific nodes (useful when you know some nodes are broken, but are still in IDLE state):

```
sbatch --exclude nodeA,nodeB
```

or via: `#SBATCH --exclude ...`

To use specific nodes:

```
sbatch --nodelist=nodeA,nodeB
```

can also use the short `-w` instead of `--nodelist`

The administrator could also define a `feature=example` in `slurm.conf` and then a user could ask for that subset of nodes via `--constraint=example`

Signal the running jobs to finish

Since each SLURM run has a limited time span, it can be configured to send a signal of choice to the program a desired amount of time before the end of the allocated time.

```
--signal=[[R][B]:]<sig_num>[@<sig_time>]
```

For the worked pattern - trap `SIGUSR1` (or similar) a few minutes before the wall-clock limit, finish the step, save a checkpoint, and exit without starting another cycle - see Sending a custom signal X minutes before the end in the fault-tolerance chapter.

Detailed job info

While most useful information is preset in various `SLURM_*` env vars, sometimes some information is missing. In such cases use:

```
scontrol show -d job $SLURM_JOB_ID
```

and then parse out what's needed.

For a job that finished its run use:

```
sacct -j JOBID
```

This command is also useful to discover if you have any `srun` jobs already running on that allocation (including those that were finished or canceled). For example, you could kill some run-away `srun` step via `scancel <jobid>.<step-id>` and you'd find that `<step-id>` via the above command. The main job will continue running if it's an interactive job even if you canceled all step jobs.

To see more details:

```
sacct --format=jobid,start,end,state,exitcode,nodelist%300 -j JOBID
sacct -j JOBID --long
```

Or to see all jobs with their sub-steps while limiting the listing to a specific partition and only for your own user:

```
sacct -u `whoami` --partition=dev  --format=jobid,start,end,state,exitcode,nodelist%300
sacct -u `whoami` --partition=prod --format=jobid,start,end,state,exitcode,nodelist%300
```

To see how a particular job was launched and all of its `srun` sub-step command lines:

```
sacct -j JOBID -o submitline -P
```

show jobs

Show only my jobs:

```
squeue -u `whoami`
```

Show jobs by job id:

```
squeue -j JOBID
```

Show jobs of a specific partition:

```
squeue --partition=dev
```

Getting information about the job

From within the slurm file one can access information about the current job's allocations.

Getting allocated hostnames and useful derivations based on that:

```
export HOSTNAMES=$(scontrol show hostnames "$SLURM_JOB_NODELIST")
export NUM_NODES=$(scontrol show hostnames "$SLURM_JOB_NODELIST" | wc -l)
export MASTER_ADDR=$(scontrol show hostnames "$SLURM_JOB_NODELIST" | head -n 1)
```

Environment variables

`#SBATCH --export=...` controls which variables from the **submission shell** are copied into the job. On stock Slurm the default is `ALL` (some sites override that via `SBATCH_EXPORT` / `SLURM_EXPORT_ENV` or a cli_filter). `SLURM_*` variables are always present regardless of the mode. Anything you `export KEY=value` inside the job script itself is also always visible to the launched program - that is independent of `--export`, which only governs what is inherited from the submit shell. The four useful forms:

1. **Full submit-shell environment (default):**

```
#SBATCH --export=ALL
```

Same as omitting `--export` on stock Slurm - writing this line alone is a no-op there. Use it only when you need to undo a site default of `NONE`, or when combining with extras as in (4).

2. **Minimal environment:**

```
#SBATCH --export=NONE
```

Drops the submit-shell environment. Useful when you want a reproducible job that does not inherit whatever happened to be exported in the login shell.

3. **Only these variables (replacement list):**

```
#SBATCH --export=KEY1,KEY2=value
```

Propagates **only** the named variables (here: current value of `KEY1`, and `KEY2` set to `value`). Everything else from the submit shell is dropped - this is not an add-on to `ALL`.

4. **Full environment plus set/override:**

```
#SBATCH --export=ALL,KEY=value
```

Keeps the full submit-shell environment **and** sets or overrides `KEY`. If you want both "everything" and an extra var at submit time, `ALL` must appear in the list - without it you get case (3).

Convert compact node list to expanded node list

Sometimes you get SLURM tools give you a string like: `node-[42,49-51]` which will require some coding to expand it into `node-42,node-49,node-50,node-51`, but there is a special tool to deal with that:

```
$ scontrol show hostnames node-[42,49-51]
node-42
node-49
node-50
node-51
```

Voila!

case study: this is for example useful if you want get a list of nodes that were drained because the job was too slow to exit, but really there is no real problem with the nodes. So this one-liner will give you the list of such nodes in an expanded format which you can then script to loop over this list to undrain these nodes after perhaps checking that the processes have died by this time:

```
sinfo -R | grep "Kill task failed" | perl -lne '/(node-.*[\d\]]+)/ && print $1' | xargs -n1
scontrol show hostnames
```

Convert SLURM_JOB_NODELIST into a hostfile

Some multi-node launchers require a `hostfile` - here is how to generate one. Expansion uses `scontrol show hostnames`, the same tool as Convert compact node list to expanded node list above, so mixed forms like `node-[42,49-51]` work:

```
# usage:
# makehostfile > hostfile
# relies on SLURM_STEP_GPUS=0,1,2... to get how many gpu slots per node
function makehostfile() {
  local slots
  slots=$(perl -F, -lane 'print scalar @F' <<<"${SLURM_STEP_GPUS:-0}")
  scontrol show hostnames "$SLURM_JOB_NODELIST" | perl -slne 'print "$_ slots=$s"' -- -s="$slots"
}
```

Aliases

Handy aliases

```
alias myjobs='squeue -u `whoami` -o "%.16i %9P %26j %.8T %.10M %.8l %.6D %.20S %R"'
alias groupjobs='squeue -u foo,bar,tar -o "%.16i %u %9P %26j %.8T %.10M %.8l %.6D %.20S %R"'
alias myjobs-pending="squeue -u `whoami` --start"
alias idle-nodes="sinfo -p prod -o '%A'"
```

Zombies

If there are any zombies left behind across nodes, send one command to kill them all.

```
srun pkill python
```

Detailed Access to SLURM Accounting

`sacct` displays accounting data for all jobs and job steps in the SLURM job accounting log or SLURM database.

So this is a great tool for analyzing past events.

For example, to see which nodes were used to run recent gpu jobs:

```
sacct -u `whoami` --partition=dev --format=jobid,start,end,state,exitcode,nodelist%300
```

`%300` here tells it to use a 300 char width for the output, so that it's not truncated.

See `man sacct` for more fields and info fields.

Queue

Cancel job

To cancel a job:

```
scancel [jobid]
```

To cancel all of your jobs:

```
scancel -u <userid>
```

To cancel all of your jobs on a specific partition:

```
scancel -u <userid> -p <partition>
```

Tips

- if you see that `salloc`'ed interactive job is scheduled to run much later than you need, try to cancel the job and ask for shorter period - often there might be a closer window for a shorter time allocation.

Logging

If we need to separate logs to different log files per node add `%N` (for short hostname) so that we have:

```
#SBATCH --output=%x-%j-%N.out
```

That way we can tell if a specific node misbehaves - e.g. has a corrupt GPU. `torchrun` can attribute the first failure to a host/rank when the entry point is wrapped with `@record`, but undecorated scripts still get a bare traceback, and a single summary does not replace having one log file per node.

Show the state of nodes

```
sinfo -p PARTITION
```

Very useful command is:

```
sinfo -s
```

and look for the main stat, e.g.:

```
NODES(A/I/O/T) "allocated/idle/other/total".
597/0/15/612
```

So here 597 out of 612 nodes are allocated. 0 idle and 15 are not available for whatever other reasons.

To check a specific partition:

```
sinfo -p gpu_p1 -o "%A"
```

gives:

```
NODES(A/I)
236/24
```

so you can see if any nodes are available on the 4x v100-32g partition (`gpu_p1`)

sinfo states

- idle: no jobs running
- alloc: nodes are allocated to jobs that are currently executing
- mix: the nodes have some of the CPUs allocated, while others are idle
- drain: the node is unavailable due to an administrative reason
- drng: the node is running a job, but will after completion not be available due to an administrative reason

Node state codes

The node state could be followed by a single character which has a special meaning. It is one of:

- `*`: The node is presently not responding and will not be allocated any new work. If the node remains non-responsive, it will be placed in the DOWN state (except in the case of COMPLETING, DRAINED, DRAINING, FAIL, FAILING nodes).
- `~`: The node is presently in powered off.
- `#`: The node is presently being powered up or configured.
- `!`: The node is pending power down.
- `%`: The node is presently being powered down.
- `$`: The node is currently in a reservation with a flag value of "maintenance".
- `@`: The node is pending reboot.
- `^`: The node reboot was issued.
- `-`: The node is planned by the backfill scheduler for a higher priority job.

Job state codes

- `CD` | Completed: The job has completed successfully.
- `CG` | Completing: The job is finishing but some processes are still active.
- `F` | Failed: The job terminated with a non-zero exit code and failed to execute.
- `PD` | Pending: The job is waiting for resource allocation. It will eventually run.
- `PR` | Preempted: The job was terminated because of preemption by another job.
- `R` | Running: The job currently is allocated to a node and is running.
- `S` | Suspended: A running job has been stopped with its cores released to other jobs.
- `ST` | Stopped: A running job has been stopped with its cores retained.

drained nodes

To see all drained nodes and the reason for drainage (edit `%50E` to make the reason field longer/shorter)

```
% sinfo -R -o "%50E %12U %19H %6t %N"
```

or just `-R` if you want it short:

```
% sinfo -R
```

Job arrays

To run a sequence of jobs, so that the next slurm job is scheduled as soon as the currently running one finishes, we use a job array.

Let's start with just 10 such jobs:

```
sbatch --array=1-10%1 array-test.slurm
```

`%1` limits the number of simultaneously running tasks from this job array to 1. Without it, it will try to run all the jobs at once, which we may want sometimes (in which case remove `%1`), but when training we need one job at a time.

Alternatively, as always this param can be part of the script:

```
#SBATCH --array=1-10%1
```

Here is toy slurm script, which can be used to see how it works:

```
#!/bin/bash
#SBATCH --job-name=array-test
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1          # crucial - only 1 task per dist per node!
#SBATCH --cpus-per-task=1            # number of cores per tasks
#SBATCH --time 00:02:00              # maximum execution time (HH:MM:SS)
#SBATCH --output=%x-%j.out           # output file name
#SBATCH --error=%x-%j.out            # error file name (same to watch just one file)
#SBATCH --partition=dev

echo $SLURM_JOB_ID
echo "I am job ${SLURM_ARRAY_JOB_ID}_${SLURM_ARRAY_TASK_ID}"
date
sleep 10
date
```

Note `$SLURM_ARRAY_JOB_ID` is the same as `$SLURM_JOB_ID`, and `$SLURM_ARRAY_TASK_ID` is the index of the job.

To see the jobs running:

```
$ squeue -u `whoami` -o "%.10i %9P %26j %.8T %.10M %.6D %.20S %R"
     JOBID PARTITION                        NAME    STATE       TIME  NODES           START_TIME
NODELIST(REASON)
591970_[2-   dev             array-test  PENDING       0:00      1  2021-07-28T20:01:06
(JobArrayTaskLimit)
```

now job 2 is running.

To cancel the whole array, cancel the job id as normal (the number before _):

```
scancel 591970
```

To cancel a specific job:

```
scancel 591970_2
```

If it's important to have the log-file contain the array id, add `%A_%a`:

```
#SBATCH --output=%x-%j.%A_%a.log
```

More details https://slurm.schedmd.com/job_array.html

Job array trains and their suspend and release

In this recipe we accomplish 2 things:

1. Allow modification to the next job's slurm script
2. Allow suspending and resuming job arrays w/o losing the place in the queue when not being ready to continue running a job

SLURM is a very unforgiving environment where a small mistake can cost days of waiting time. But there are strategies to mitigate some of this harshness.

SLURM jobs have a concept of "age" in the queue which besides project priority governs when a job gets scheduled to run. If your have just scheduled a new job it has no "age" and will normally be put to run last compared to jobs that have entered the queue earlier. Unless of course this new job comes from a high priority project in which case it'll progress faster.

So here is how one can keep the "age" and not lose it when needing to fix something in the running script or for example to switch over to another script.

The idea is this:

1. `sbatch` a long job array, e.g., `-array=1-50%1`
2. inside the slurm script don't have any code other than `source another-script.slurm` - so now you can modify the target script or symlink to another script before the next job starts
3. if you need to stop the job array train - don't cancel it, but put it on hold without losing your place in a queue
4. when ready to continue - release the job array - only the time while it was on hold is not counted towards its age, but all the previous age is retained.

The number of nodes, time and hardware and partition of a running job cannot be modified, but you can change pending jobs in the job array by `scontrol update jobid=<desired_job_id> numnodes=<new number> partition=<new partition>`.

If you do have `sudo` access then you can change the job time of the current job as well.

Here is an example:

Create a job script:

```
$ cat train-64n.slurm
#!/bin/bash
#SBATCH --job-name=tr8-104B
#SBATCH --nodes=64
#SBATCH --ntasks-per-node=1         # crucial - only 1 task per dist per node!
#SBATCH --cpus-per-task=96          # number of cores per tasks
#SBATCH --gres=gpu:8                # number of gpus
#SBATCH --time 20:00:00             # maximum execution time (HH:MM:SS)
#SBATCH --output=%x-%j.out          # output file name
#SBATCH --partition=dev

source tr8-104B-64.slurm
```

Start it as:

```
sbatch --array=1-50%1 train-64.slurm
```

Now you can easily edit `tr8-104B-64.slurm` before the next job run and either let the current job finish if it's desired or if you need to abort it, just kill the currently running job, e.g. `1557903_5` (not job array `1557903`) and have the train pick up where it left, but with the edited script.

The nice thing is that this requires no changes to the original script (`tr8-104B-64.slurm` in this example), and the latter can still be started on its own.

Now, what if something is wrong and you need 10min or 10h to fix something. In this case we put the train on hold using:

```
scontrol hold <jobid>
```

with being either a "normal" job, the id of a job array or the id for a job array step

and then when ready to continue release the job:

```
scontrol release <jobid>
```

How to keep an salloc allocation alive while exiting its shell

If you run allocated a node like so:

```
salloc --partition=dev --nodes=1 --ntasks-per-node=1 --time=1:00:00 bash
```

and you exited the shell, or your ssh connection got dropped, the allocation will be lost.

If you want to open an allocation that should survive exiting the shell, use `--no-shell` and no `bash` like so:

```
salloc --no-shell --partition=dev --nodes=1 --ntasks-per-node=1 --time=1:00:00
```

and now if you need to join the session see How to rejoin the allocated node interactively.

But beware, that if you `ssh` to the allocated node and launch something normally and then close the connection that job will be lost as the connecting shell will send `SIGHUP` to its child processes. To avoid that and to keep the job running use `nohup` while putting the program into a background process. Example:

```
nohup my-program &
```

`nohup` will ignore `SIGHUP` and will redirect stderr to stdout and append stdout to a special file `nohup.out`. If you want to control where the std streams should be written use normal stdout redirect > or >>, e.g.:

```
nohup my-program >> some-file.txt &
```

As mentioned earlier the program is also sent into the background with `&`.

Now you can safely disconnect and the program will continue running when you come back.

This solution will prevent the program from exiting, but you won't be able to interact with it normally when you connect again as the std streams will be redirected. You can of course still kill the program via its pid, chance its `nice` state, etc., like you'd do with any other process.

But if you want to use something where you can disconnect and reconnect and continue using the program normally you'd have to use a terminal multiplexer program like `tmux` or GNU `screen` which run a daemon on the node and allow you to regain the normal control over the program on reconnection. There are also `mosh` and other similar tools which further aid this process.

How to rejoin the allocated node interactively

To have multiple interactive shells into the same job `--overlap` should be used.

For example, in console A, let's allocate a single node:

```
$ salloc --partition=dev --nodes=1 --ntasks-per-node=1 --cpus-per-task=26 --gres=gpu:1
--time=2:00:00 bash
salloc: Granted job allocation 1916
salloc: Nodes my-node-1 are ready for job
```

In console B:

```
$ srun --overlap --pty --jobid 1916 bash
```

and the above can be repeated in as many consoles as wanted.

If it's the first pseudo terminal shell you don't even need `--overlap`, but you need it for the additional shells.

It works the same if you initially allocated the node via `srun --pty`

```
srun --pty -p dev --gpus 8 --time=2:00:00 bash
```

You can, of course, also access the node via `ssh` but if your SLURM has been setup to do all kinds of virtualizations (e.g. give only a few GPUs to each user, or virtualize `/tmp/` or `/scratch` with auto-cleanup on exit), the view from `ssh` won't be the same. For example, if a job allocated 2 GPUs, the ssh shell will show all of the GPUs and not just the 2 - so if you're sharing the node with others this won't work well.

This works for multi-node allocations and by default you will get an interactive shell on the first node of the allocation. If you want to enter a specific node use `-w` to specify it. For example, say you got `node-[1-4]` allocated and you want to enter `node-3`, then specify:

```
srun --pty -p dev --gpus 8 --time=2:00:00 -w node-3 bash
```

and if it fails with:

```
srun: error: Unable to create step for job 1930: Invalid generic resource (gres) specification
```

add back the `--gres=gpu:8` setting. You won't need to do it if your original allocation command used this flag already.

Troubleshooting

`SLURM_PROCID` early interpolation

When using SLURM with multi-node setup it's crucial that this is set correctly:

```
"--machine_rank \$SLURM_PROCID"
```

it must not be interpolated before time, since if this is set as `"--machine_rank $SLURM_PROCID"` the launcher will hang.

It's best to isolate the launcher from the program like so:

```
export MASTER_ADDR=$(scontrol show hostnames $SLURM_JOB_NODELIST | head -n 1)
export MASTER_PORT=3333
ACCELERATE_CONFIG_FILE=path/to/accelerate.config.yaml # edit me
LAUNCHER="python -u -m accelerate.commands.launch \
    --rdzv_conf "rdzv_backend=c10d,rdzv_endpoint=$MASTER_ADDR:$MASTER_PORT" \
    --config_file $ACCELERATE_CONFIG_FILE \
    --main_process_ip $MASTER_ADDR \
    --main_process_port $MASTER_PORT \
    --machine_rank \$SLURM_PROCID \
    --role \$(hostname -s|tr -dc '0-9'): --tee 3 \
    "
PROGRAM="myprogram.py"

CMD="$LAUNCHER $PROGRAM"

SRUN_ARGS=" \
    --wait=60 \
    --kill-on-bad-exit=1 \
    --unbuffered \
    --jobid $SLURM_JOBID \
    "

srun $SRUN_ARGS bash -c "$CMD" 2>&1 | tee -a main_log.txt
```

Now the launcher will always work and the users will only need to tweak the `PROGRAM` variable.

With `torchrun`:

```
GPUS_PER_NODE=8
NNODES=$SLURM_NNODES
export MASTER_ADDR=$(scontrol show hostnames $SLURM_JOB_NODELIST | head -n 1)
export MASTER_PORT=3333
# note `\$SLURM_PROCID` and `\$(hostname ...)` - interpolate at `srun` time, not here
LAUNCHER="python -u -m torch.distributed.run \
```

```
    --nproc_per_node $GPUS_PER_NODE \
    --nnodes $NNODES \
    --node_rank \$SLURM_PROCID \
    --rdzv_endpoint $MASTER_ADDR:$MASTER_PORT \
    --rdzv_backend c10d \
    --max_restarts 0 \
    --role \$(hostname -s|tr -dc '0-9'): \
    --tee 3 \
    "
```

See Single and Multi-node Launchers with SLURM for complete working examples.

Mismatching nodes number

If the pytorch launcher fails it often means that the number of SLURM nodes and the launcher nodes are mismatching, e.g.:

```
grep -ir nodes= tr123-test.slurm
#SBATCH --nodes=40
NNODES=64
```

This won't work. They have to match.

You can add a sanity check to your script:

```
#!/bin/bash
#SBATCH --job-name=test-mismatch
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=1          # crucial - only 1 task per dist per node!
#SBATCH --cpus-per-task=96           # number of cores per tasks
#SBATCH --gres=gpu:8                 # number of gpus
#SBATCH --time 0:05:00               # maximum execution time (HH:MM:SS)
#SBATCH --output=%x-%j.out           # output file name
#SBATCH --partition=prod

[...]

NNODES=2

# sanity check for having NNODES and `#SBATCH --nodes` match, assuming you use NNODES variable
if [ "$NNODES" != "$SLURM_NNODES" ]; then
    echo "Misconfigured script: NNODES=$NNODES != SLURM_NNODES=$SLURM_NNODES"
    exit 1
fi

[...]
```

or you could just do:

```
#SBATCH --nodes=2
[...]
NNODES=$SLURM_NNODES
```

and then it will always be correct

Find faulty nodes and exclude them

Sometimes a node is broken, which prevents one from training, especially since restarting the job often hits the same set of nodes. So one needs to be able to isolate the bad node(s) and exclude it from `sbatch`.

To find a faulty node, write a small script that reports back the status of the desired check.

For example to test if cuda is available on all nodes:

```
python -c 'import torch, socket; print(f"{socket.gethostname()}: {torch.cuda.is_available()}")'
```

and to only report the nodes that fail:

```
python -c 'import torch, socket; torch.cuda.is_available() or print(f"Broken node:
{socket.gethostname()}") '
```

Of course, the issue could be different - e.g. gpu can't allocate memory, so change the test script to do a small allocation on cuda. Here is one way:

```
python -c "import torch; torch.ones(1000,1000).cuda()"
```

But since we need to run the test script on all nodes and not just the first node, the slurm script needs to run it via `srun`. So our first diagnostics script can be written as:

```
srun --jobid $SLURM_JOBID bash -c 'python -c "import torch, socket; print(socket.gethostname(),
torch.cuda.is_available())"'
```

I slightly changed it, due to an issue with quotes.

You can always convert the one liner into a real script and then there is no issue with quotes.

```
$ cat << EOT >> test-nodes.py
#!/usr/bin/env python
import torch, socket
print(socket.gethostname(), torch.cuda.is_available())
EOT
$ chmod a+x ./test-nodes.py
```

Now let's create a driver slurm script. Use a few minutes time for this test so that SLURM yields it faster:

```
#!/bin/bash
#SBATCH --job-name=test-nodes
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=1          # crucial - only 1 task per dist per node!
#SBATCH --cpus-per-task=96           # number of cores per tasks
#SBATCH --gres=gpu:8                 # number of gpus
#SBATCH --time 0:05:00               # maximum execution time (HH:MM:SS)
#SBATCH --output=%x-%j.out           # output file name
#SBATCH --partition=prod

source $six_ALL_CCFRWORK/start-prod   # edit me - site-specific env setup
srun --jobid $SLURM_JOBID ./test-nodes.py
```

Once it runs check the logs to see if any reported `False`, those are the nodes you want to exclude.

Now once the faulty node(s) is found, feed it to `sbatch`:

```
sbatch --exclude=hostname1,hostname2 ...
```

and `sbatch` will exclude the bad nodes from the allocation.

Here are a few more situations and how to find the bad nodes in those cases:

Broken NCCL

If you're testing something that requires distributed setup, it's a bit more complex. Here is a slurm script that tests that NCCL works. It sets up NCCL and checks that barrier works:

```
#!/bin/bash
#SBATCH --job-name=test-nodes-nccl
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=1          # crucial - only 1 task per dist per node!
#SBATCH --cpus-per-task=96           # number of cores per tasks
#SBATCH --gres=gpu:8                 # number of gpus
#SBATCH --time 0:05:00               # maximum execution time (HH:MM:SS)
#SBATCH --output=%x-%j.out           # output file name
#SBATCH --partition=prod

source $six_ALL_CCFRWORK/start-prod   # edit me - site-specific env setup

NNODES=2

GPUS_PER_NODE=4
MASTER_ADDR=$(scontrol show hostnames $SLURM_JOB_NODELIST | head -n 1)
MASTER_PORT=6000

export LAUNCHER="python -u -m torch.distributed.run \
    --nproc_per_node $GPUS_PER_NODE \
    --nnodes $NNODES \
```

```
    --master_addr $MASTER_ADDR \
    --master_port $MASTER_PORT \
    "

export SCRIPT=test-nodes-nccl.py

cat << EOT > $SCRIPT
#!/usr/bin/env python
import torch.distributed as dist
import torch
import socket
import os
import fcntl

def printflock(*msgs):
    """ print """
    with open(__file__, "r") as fh:
        fcntl.flock(fh, fcntl.LOCK_EX)
        try:
            print(*msgs)
        finally:
            fcntl.flock(fh, fcntl.LOCK_UN)

local_rank = int(os.environ["LOCAL_RANK"])
torch.cuda.set_device(local_rank)
dist.init_process_group("nccl")
header = f"{socket.gethostname()}-{local_rank}"
try:
    dist.barrier()
    printflock(f"{header}: NCCL {torch.cuda.nccl.version()} is OK")
except:
    printflock(f"{header}: NCCL {torch.cuda.nccl.version()} is broken")
    raise
EOT

echo $LAUNCHER --node_rank $SLURM_PROCID $SCRIPT

srun --jobid $SLURM_JOBID bash -c '$LAUNCHER --node_rank $SLURM_PROCID $SCRIPT'
```

The script uses `printflock` to solve the interleaved print outputs issue.

GPU Memory Check

This tests if each GPU on the allocated nodes can successfully allocate 77GiB (e.g. to test 80GB A100s) (have to subtract a few GiBs for cuda kernels).

```
import torch, os
import time
import socket
hostname = socket.gethostname()
```

```
local_rank = int(os.environ["LOCAL_RANK"]);

gbs = 77
try:
    torch.ones((gbs*2**28)).cuda(local_rank).contiguous() # alloc on cpu, then move to gpu
    print(f"{local_rank} {hostname} is OK")
except:
    print(f"{local_rank} {hostname} failed to allocate {gbs}GiB of accelerator memory")
    pass

time.sleep(5)
```

Broken Network

Yet another issue with a node is when its network is broken and other nodes fail to connect to it.

You're likely to experience it with an error similar to:

```
work = default_pg.barrier(opts=opts)
RuntimeError: NCCL error in: /opt/conda/conda-bld/pytorch_1616554793803/work/torch/lib/c10d/
ProcessGroupNCCL.cpp:825, unhandled system error, NCCL version 2.7.8
ncclSystemError: System call (socket, malloc, munmap, etc) failed.
```

Here is how to debug this issue:

1. Add:

```
export NCCL_DEBUG=INFO
```

before the `srun` command and re-run your slurm script.

2. Now study the logs. If you find:

```
r11i6n2:486514:486651 [1] include/socket.h:403 NCCL WARN Connect to 10.148.3.247<56821> failed :
Connection refused
```

Let's see which node refuses to accept connections. We get the IP address from the error above and reverse resolve it to its name:

```
nslookup 10.148.3.247
247.3.148.10.in-addr.arpa       name = r10i6n5.ib0.xa.idris.fr.
```

Add `--exclude=r10i6n5` to your `sbatch` command and report the faulty node to your cluster's admins.

Run py-spy or any other monitor program across all nodes

When dealing with hanging, here is how to automatically log `py-spy` traces for each process.

Of course, this same process can be used to run some command for all nodes of a given job. i.e. it can be used to run something during the normal run - e.g. dump all the memory usage in each process via `nvidia-smi` or whatever other program is needed to be run.

```
cd ~/prod/code/tr8b-104B/bigscience/train/tr11-200B-ml/

salloc --partition=prod --nodes=40 --ntasks-per-node=1 --cpus-per-task=96 --gres=gpu:8 --time
20:00:00

bash 200B-n40-bf16-mono.slurm
```

In another shell get the JOBID for the above `salloc`:

```
squeue -u `whoami` -o "%.16i %9P %26j %.8T %.10M %.8l %.6D %.20S %R"
```

adjust jobid per above and the nodes count:

```
srun --jobid=2180718 --gres=gpu:0 --nodes=40 --tasks-per-node=1 --output=trace-%N.out sh -c 'ps
aux | grep python | egrep -v "grep|srun" | grep `whoami` | awk "{print \$2}" | xargs -I {} py-spy
dump --native --pid {}' || echo "failed"
```

though most likely `--nodes=40` is redundant since `salloc` already knows the number of nodes.

Now all `py-spy` traces go into the `trace-$nodename.out` files under `cwd`.

The key is to use `--gres=gpu:0` or otherwise the 2nd `srun` will block waiting for the first one to release the gpus.

Also the assumption is that some conda env that has `py-spy` installed got activated in `~/.bashrc`. If yours doesn't already do that, add the instruction to load the env to the above command, before the `py-spy` command - it'll fail to find it otherwise.

Don't forget to manually release the allocation when this process is done.

Crontab Emulation

One of the most important Unix tools is the crontab, which is essential for being able to schedule various jobs. It however usually is absent from SLURM environment. Therefore one must emulate it. Here is how.

For this presentation we are going to use `$WORK/cron/` as the base directory, assuming you have an exported environment variable `WORK` pointing to some location on your filesystem - if you use Bash you can set it up in your `~/.bash_profile` or if a different shell is used use whatever startup equivalent file is.

1. A self-perpetuating scheduler job

We will use `$WORK/cron/scheduler` dir for scheduler jobs, `$WORK/cron/cron.daily` for daily jobs and `$WORK/cron/cron.hourly` for hourly jobs:

```
$ mkdir -p $WORK/cron/scheduler
$ mkdir -p $WORK/cron/cron.daily
$ mkdir -p $WORK/cron/cron.hourly
```

Now copy these two slurm script in `$WORK/cron/scheduler`:

- cron-daily.slurm
- cron-hourly.slurm

after editing those to fit your specific environment's account and partition information.

Now you can launch the crontab scheduler jobs:

```
$ cd $WORK/cron/scheduler
$ sbatch cron-hourly.slurm
$ sbatch cron-daily.slurm
```

This is it, these jobs will now self-perpetuate and usually you don't need to think about it again unless there is an event that makes SLURM lose all its jobs.

2. Daily and Hourly Cronjobs

Now whenever you want some job to run once a day, you simply create a slurm job and put it into the `$WORK/cron/cron.daily` dir.

Here is an example job that runs daily to rebuild a file index of `$WORK` with `updatedb` (on current distributions that is usually `plocate`'s `updatedb`; older systems had `mlocate`):

```
$ cat $WORK/cron/cron.daily/mlocate-update.slurm
#!/bin/bash
#SBATCH --job-name=mlocate-update    # job name
#SBATCH --ntasks=1                   # number of MP tasks
#SBATCH --nodes=1
#SBATCH --hint=nomultithread         # we get physical cores not logical
#SBATCH --time=1:00:00               # maximum execution time (HH:MM:SS)
#SBATCH --output=%x-%j.out           # output file name
#SBATCH --partition=PARTITION     # edit me
#SBATCH --account=GROUP@PARTITION # edit me

set -e
date
echo "updating file index db"
/usr/bin/updatedb -o $WORK/lib/mlocate/work.db -U $WORK --require-visibility 0
```

This builds an index of the files under `$WORK` which you can then quickly query with:

```
/usr/bin/locate -d $WORK/lib/mlocate/work.db pattern
```

To stop running this job, just move it out of the `$WORK/cron/cron.daily` dir.

The same principle applies to jobs placed into the `$WORK/cron/cron.hourly` dir. These are useful for running something every hour.

Please note that this crontab implementation is approximate timing-wise, due to various delays in SLURM scheduling they will run approximately every hour and every day. You can recode these to ask SLURM to start something at a more precise time if you have to, but most of the time the just presented method works fine.

Additionally, you can code your own variations to meet specific needs of your project, e.g., every-30min or every-12h jobs.

3. Cleanup

Finally, since every cron launcher job will leave behind a log file (which is useful if for some reason things don't work), you want to create a cronjob to clean up these logs. Otherwise you may run out of inodes - these logs files are tiny, but there could be tens of thousands of those.

You could use something like this in a daily job.

```
find $WORK/cron -name "*.out" -mtime +7 -exec rm -f {} +
```

Please note that it's set to only delete files that are older than 7 days, in case you need the latest logs for diagnostics.

Nuances

The scheduler runs with Unix permissions of the person who launched the SLURM cron scheduler job and so all other SLURM scripts launched by that cron job.

Self-perpetuating SLURM jobs

The same approach used in building a scheduler can be used for creating stand-alone self-perpetuating jobs.

For example:

```
#!/bin/bash
#SBATCH --job-name=watchdog          # job name
#SBATCH --ntasks=1                   # number of MP tasks
#SBATCH --nodes=1
#SBATCH --time=0:30:00               # maximum execution time (HH:MM:SS)
#SBATCH --output=%x-%j.out           # output file name
#SBATCH --partition=PARTITION        # edit me

# ensure to restart self first 1h from now
RUN_FREQUENCY_IN_HOURS=1
sbatch --begin=now+${RUN_FREQUENCY_IN_HOURS}hour watchdog.slurm

... do the watchdog work here ...
```

and you launch it once with:

```
sbatch watchdog.slurm
```

This then will immediately schedule itself to be run 1 hour from the launch time and then the normal job work will be done. Regardless of whether the rest of the job will succeed or fail, this job will continue relaunching itself approximately once an hour. This is imprecise due to scheduler job starting overhead and node availability issues. But if there is a least one spare node available and the job itself is quick to finish the requirement to run at an approximate frequency should be sufficient.

As the majority of SLURM environment in addition to the expensive GPU nodes also provide much cheaper CPU-only nodes, you should choose a CPU-only SLURM partition for any jobs that don't require GPUs to run.

Overcoming the lack of group SLURM job ownership

SLURM runs on Unix, but surprisingly its designers haven't adopted the concept of group ownership with regards to SLURM jobs. So if a member of your team started an array of 10 jobs 20h each, and went on vacation - unless you have `sudo` access you now can't do anything to stop those jobs if something is wrong.

I'm yet to find why this is so, but so far we have been using a kill switch workaround. You have to code it in your framework. For example, see how it was implemented in Megatron-DeepSpeed (Meg-DS). The program polls for a pre-configured at start up path on the filesystem and if it finds a file there, it exits.

So if we start Meg-DS with `--kill-switch-path $WORK/tmp/training17-kill-switch` and then at any point we need to kill the SLURM job, we simply do:

```
touch $WORK/tmp/training17-kill-switch
```

and the next time the program gets to check for this file it'll detect the event and will exit voluntarily. If you have a job array, well, you will have to wait until each job starts, detects the kill switch and exits.

Of course, don't forget to remove it when you're done stopping the jobs.

```
rm $WORK/tmp/training17-kill-switch
```

Now, this doesn't always work. If the job is hanging, it'll never come to the point of checking for kill-switch and the only solution here is to contact the sysadmins to kill the job for you. Sometimes if the hanging is a simple case pytorch's distributed setup will typically auto-exit after 10min of preset timeout time, but it doesn't always work.

How to gracefully exit on SLURM job preemption

There are several ways to gracefully handle time- and QoS-based SLURM pre-emption which are covered indepth in this section: Dealing with forced job preemption.

How many GPUs a job uses

To figure out how many gpus are used by an already running job, parse the `JOB_GRES=gpu:` entry in `show job -d` output. For example, if the job was started with:

```
srun --pty --partition=dev --nodes=2 --ntasks-per-node=1 --gres=gpu:8 --time=8:00:00 bash
```

that is we allocated 16 GPUs, we can now get that number back programmatically via:

```
$ TOTAL_JOB_GPUS=$(scontrol show job -d $SLURM_JOBID | perl -ne 'm|JOB_GRES=gpu:(\d+)| && print
$1')
$ echo $TOTAL_JOB_GPUS
16
```

Replace `$SLURM_JOBID` with the SLURM job id if it's not already set in the shell you run the command from (squeue).

How long did the job take to run

While normally `squeue` will show you the duration of the currently running job, in order to see how long a job run for when it finished, you need to know the job id and then you can query it like so:

```
$ sacct -j 22171 --format=JobID,JobName,State,Elapsed
JobID           JobName      State    Elapsed
------------ ---------- ---------- ----------
22171          example   COMPLETED   00:01:49
```

so we know the job finished running in under 2min.

FairShare

Many SLURM clusters use the FairShare system where the more someone uses the cluster the less of the priority they get to run jobs or if there is a pre-emption in place they are more likely to get pre-empted

To see your FairShare scores run:

```
sshare
```

Example:

```
Account                  User  RawShares  NormShares    RawUsage  EffectvUsage  FairShare
-------------------- ---------- ---------- ----------- ----------- ------------- ----------
root                                          0.000000   711506073      1.000000
 all                                    1    0.500000   711506073      1.000000
  all                       stas          1    0.022727    14106989      0.019827   0.288889
```

If your FairShare score is more than 0.5 that means you have been using the cluster less than what you have been allocated, if it's less than 0.5 it means you have been using more than what was allocated.

As the time passes this score gets decayed so if you were having a very low score and have you have been using the cluster much less then your score will raise over time.

To see the score of a specific user:

```
sshare -u username
```

To see everybody's scores, sorted by FairShare:

```
sshare --all | sort -nk7 -r
```

This is the most important output, since it doesn't really matter what your score is alone. What matters is your score relative to all other users. Everybody who has a higher score than you will have a higher chance at getting they job yielded first and a lower chance of getting their job preempted.

Besides FairShare the priorities are typically configured based on a combination of multiple metrics, usually including the length of time a job has been waiting in the queue, job size, Quality of Service (QOS) setting, partition specifics, etc. The specifics will depend on how the slurm has been configured by your sysadmin.

SLURM Performance

Here you will find discussions of SLURM-specific settings that impact performance.

srun's `--cpus-per-task` may need to be explicit

You need to make sure that the launched by `srun` program receives as many cpu-cores as intended. For example, in a typical case of a ML training program, each gpu needs at least one cpu-core for the process driving it plus a few more cores for the `DataLoader`. You need multiple cores so that each task can be performed in parallel. If you have 8 gpus and 2 `DataLoader` workers per gpu, you need at least `3*8=24` cpu-cores per node.

The number of cpus per task is defined by `--cpus-per-task`, which is passed to `sbatch` or `salloc` and originally `srun` would inherit this setting. However, recently this behavior has changed:

A quote from the `sbatch` manpage:

> NOTE: Beginning with 22.05, srun will not inherit the --cpus-per-task value requested by salloc or sbatch. It must be requested again with the call to srun or set with the SRUN_CPUS_PER_TASK environment variable if desired for the task(s).

Which means that if in the past your SLURM script could have been:

```
#SBATCH --cpus-per-task=48
[...]

srun myprogram
```

and the program launched by `srun` would have received 48 cpu-cores because `srun` used to inherit the `--cpus-per-task=48` settings from `sbatch` or `salloc` settings, according to the quoted documentation this behavior changed somewhere in the SLURM 22.05 series and is definitely no longer true as of the 23.x series.

footnote: I tested with SLURM@22.05.09 and the old behavior was still true, but this is definitely the case with 23.x series. So the change might have happened in the later 22.05 series.

So if you leave things as is, now the program will receive just 1 cpu-core (unless the `srun` default has been modified).

You can easily test if your SLURM setup is affected, using `os.sched_getaffinity(0)`, as it shows which cpu-cores are eligible to be used by the current process. So it should be easy to count those with `len(os.sched_getaffinity(0))`.

Here is how you can test if you're affected:

```
$ cat test.slurm
#!/bin/bash
#SBATCH --job-name=test-cpu-cores-per-task
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=48   # adapt to your env if you have less than 48 cpu cores
#SBATCH --time=0:10:00
```

```
#SBATCH --partition=x        # adapt to your env to the right partition name
#SBATCH --output=%x-%j.out

srun python -c 'import os; print(f"visible cpu cores: {len(os.sched_getaffinity(0))}")'
```

If you get

```
visible cpu cores: 48
```

then you don't need to do anything, if however you get:

```
visible cpu cores: 1
```

or another value smaller than 48 then you're affected.

To fix that you need to change your SLURM script to either:

```
#SBATCH --cpus-per-task=48
[...]

srun --cpus-per-task=48 myprogram
```

or:

```
#SBATCH --cpus-per-task=48
[...]

export SRUN_CPUS_PER_TASK=48
srun myprogram
```

or automate it with write-once-and-forget:

```
#SBATCH --cpus-per-task=48
[...]

export SRUN_CPUS_PER_TASK=$SLURM_CPUS_PER_TASK
srun myprogram
```

To enable Hyper-Threads or not

As explained in the Hyper-Threads section you should be able to double the number of available cpu-cores if your CPUs support hyper-threading and for some workloads this may lead to an overall faster performance.

However, you should test the performance w/ and w/o HT, compare the results and choose the setting that gives the best

outcome.

case study: on AWS p4 nodes I discovered that enabling HT made the network throughput 4x slower. Since then we were careful to have HT disabled on that particular setup.

Single and Multi-Node Launchers with SLURM

The following are complete SLURM scripts that demonstrate how to integrate various launchers with software that uses `torch.distributed` (but should be easily adaptable to other distributed environments).

- torchrun - to be used with PyTorch distributed.
- accelerate - to be used with HF Accelerate.
- lightning - to be used with Lightning ("PyTorch Lightning" and "Lightning Fabric").
- srun - to be used with the native SLURM launcher - here we have to manually preset env vars that `torch.distributed` expects.

All of these scripts use torch-distributed-gpu-test.py as the demo script, which you can copy here with just:

```
cp ../../../debug/torch-distributed-gpu-test.py .
```

assuming you cloned this repo. But you can replace it with anything else you need.

Kubernetes

In general IMHO kubernetes is the wrong environment for doing training work, with SLURM being a much more user-friendly orchestration. SLURM builds on top of Unix and so it just makes multi-node coordination easier. But k8s reinvents the wheel completely, leading to lots of complexities and many normal Unix features are missing from its feature set, making it into a very challenging environment to work in, requiring a large support team to just deal with so many k8s problems and a lack of rudimentary Unix features (e.g., you can't even have Unix usernames and all users run under the same Unix username - think accountability and security!).

When forced into k8s one will find a way to get their work done, but at a huge human and $$ cost to the training shop.

This is going to be a small chapter since I can't imagine how I'd even approach covering k8s complexities here, so I'm just going to cover important gotchas that should ease your work a bit.

Setup

Overcoming job reset on CPU OOM event

This default feature enabled in k8s v1.28 makes absolutely no sense in the context of interactive training jobs. If any of your processes get CPU OOM'ed you get kicked out and your job gets reset. This feature is good for serving inference in production, but it's a huge problem for interactive training work, for example when one is trying to find out an optimal performance training configuration, as instead of being able to recover from a crashed process, adjust things and try again, you have to start from scratch and not even have logs, leading to the problem unless you manually sync those in time.

So to overcome this you need to get k8s to set `memory.oom.group = 0` either on the node pool level or cluster level configs - the default is `memory.oom.group = 1` so if your job gets killed on a cpu-oom event ask your k8s admin to make this change. The value of `0` will just kill the process that caused cpu-oom.

Here is how this is done. Kubernetes 1.32 introduced the kubelet flag `singleProcessOOMKill`, which allows you to set `memory.oom.group = 0`.

```
compute:
  additionalNodePools:
    - name: foo
      kubeletConfig:
        singleProcessOOMKill: true
```

To check the actual setting from within the running node, do:

```
$ cat /sys/fs/cgroup/memory.oom.group
```

A value of `0` means the fix is in effect and only the offending process will be killed on a CPU OOM event; a value of `1` is the problematic default that kills the whole job's process group.

Training

Subsections:

- Model parallelism
- Performance
- Fault Tolerance
- Reproducibility
- Instabilities
- Checkpoints
- Training hyper-parameters and model initializations
- Tensor precision / Data types
- Emulate a multi-node setup using just a single node - instructions on how to emulate a multi-node setup using just a single node - we use the `deepspeed` launcher here.
- Re-train HF hub models from scratch using finetuning examples
- Datasets

Tools:

- printflock.py - a tiny library that makes your `print` calls non-interleaved in a multi-gpu environment.
- multi-gpu-non-interleaved-print.py - a `flock`-based wrapper around `print` that prevents messages from getting interleaved when multiple processes print at the same time - which is the case with `torch.distributed` used with multiple-gpus.

Model Parallelism

Parallelism overview

In the modern Machine Learning the various approaches to parallelism are used to:

1. Overcome GPU memory limitations. Examples:
    - fit very large models - e.g., t5-11b is 45GB in just model params
    - fit very long sequences - e.g., 128K context length for Llama 3.1 models
2. significantly speed up training - finish training that would take a year in hours

We will first discuss in depth various 1D parallelism techniques and their pros and cons and then look at how they can be combined into 2D and 3D parallelism to enable an even faster training and to support even bigger models. Various other powerful alternative approaches will be presented.

While the main concepts most likely will apply to any other framework, this article is focused on PyTorch-based implementations.

Two main approaches are used to enable training and inferring models that are bigger than the accelerator's memory:

1. 3D parallelism - very network efficient, but can be very invasive into the modeling code and require a lot more work to make it work correctly
2. ZeRO parallelism - not very network efficient, but requires close to zero changes to the modeling code and very easy to make to work.

Scalability concepts

The following is the brief description of the main concepts that will be described later in depth in this document.

1. Data Parallelism (DP) - the same setup is replicated multiple times, and each being fed a slice of the data. The processing is done in parallel and all setups are synchronized at the end of each training step.
2. TensorParallelism (TP) - each tensor is split up into multiple chunks, so instead of having the whole tensor reside on a single gpu, each shard of the tensor resides on its designated gpu. During forward and backward processing the shards execute concurrently and exchange partial activations or gradients at layer-specific boundaries, often using all-reduce, all-gather, or reduce-scatter rather than synchronizing only at the end of the step. This is what one may call horizontal parallelism, as the splitting happens on horizontal level. See the PyTorch Tensor Parallel tutorial.
3. PipelineParallelism (PP) - the model is split up vertically (layer-level) across multiple GPUs, so that only one or several layers of the model are places on a single gpu. Each gpu processes in parallel different stages of the pipeline and working on a small chunk of the batch.
4. Zero Redundancy Optimizer (ZeRO) - Also performs sharding of the tensors somewhat similar to TP, except the whole tensor gets reconstructed in time for a forward or backward computation, therefore the model doesn't need to be modified. It also supports various offloading techniques to compensate for limited GPU memory. Sharded DDP is another name for the foundational ZeRO concept as used by various other implementations of ZeRO.
5. Sequence Parallelism - training on long input sequences requires huge amounts of GPU memory. This technique splits the processing of a single sequence across multiple GPUs.
6. Expert Parallelism - Mixture-Of-Experts (MoE) can be partitioned so that each expert has a dedicated GPU (or several of them).

The introduction sections of this paper is probably one of the best explanations I have found on most common parallelism techniques Breadth-First Pipeline Parallelism.

Data Parallelism

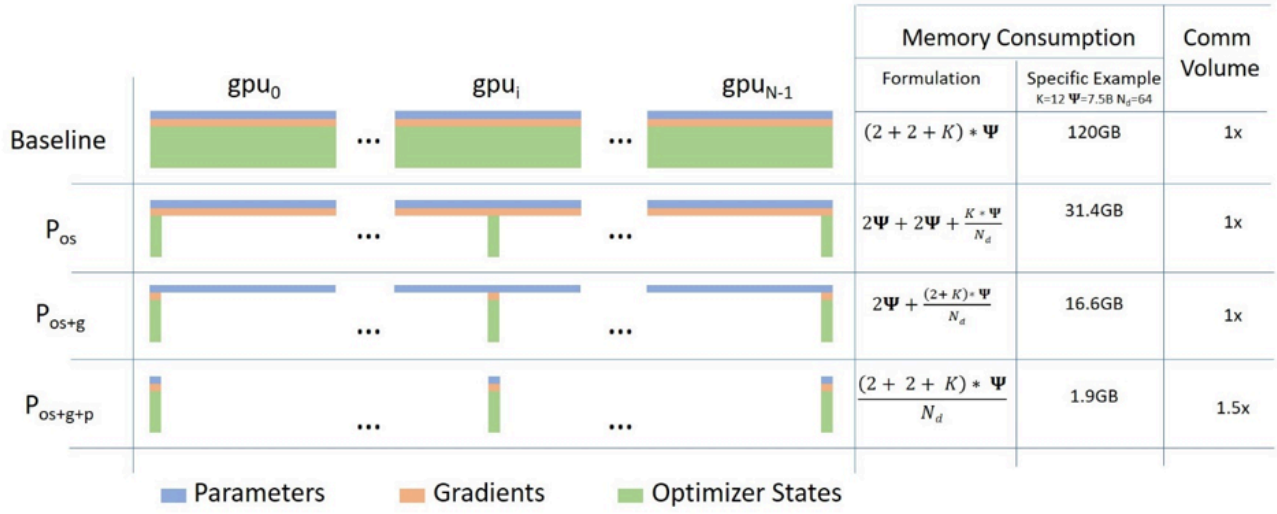
DDP

Most users with just 2 GPUs already enjoy the increased training speed up thanks to `DataParallel` (DP) and `DistributedDataParallel` (DDP) that are almost trivial to use. This is a built-in feature of PyTorch.

For details see DistributedDataParallel

ZeRO Data Parallelism

ZeRO-powered data parallelism (ZeRO-DP) is described on the following diagram from this blog post



It can be difficult to wrap one's head around it, but in reality the concept is quite simple. This is just the usual `DataParallel` (DP), except, instead of replicating the full model params, gradients and optimizer states, each GPU stores only a slice of it. And then at run-time when the full layer params are needed just for the given layer, all GPUs synchronize to give each other parts that they miss - this is it.

Consider this simple model with 3 layers, where each layer has 3 params:

```
La | Lb | Lc
---|----|---
a0 | b0 | c0
a1 | b1 | c1
a2 | b2 | c2
```

Layer La has weights a0, a1 and a2.

If we have 3 GPUs, the Sharded DDP (= ZeRO-DP) splits the model onto 3 GPUs like so:

```
GPU0:
La | Lb | Lc
---|----|---
a0 | b0 | c0

GPU1:
```

```
La | Lb | Lc
---|----|---
a1 | b1 | c1

GPU2:
La | Lb | Lc
---|----|---
a2 | b2 | c2
```

In a way this is the same horizontal slicing, as tensor parallelism, if you imagine the typical DNN diagram. Vertical slicing is where one puts whole layer-groups on different GPUs. But it's just the starting point.

Now each of these GPUs will get the usual mini-batch as it works in DP:

```
x0 => GPU0
x1 => GPU1
x2 => GPU2
```

The inputs are unmodified - they think they are going to be processed by the normal model.

First, the inputs hit the layer La.

Let's focus just on GPU0: x0 needs a0, a1, a2 params to do its forward path, but GPU0 has only a0 - it gets sent a1 from GPU1 and a2 from GPU2, bringing all pieces of the model together.

In parallel, GPU1 gets mini-batch x1 and it only has a1, but needs a0 and a2 params, so it gets those from GPU0 and GPU2.

Same happens to GPU2 that gets input x2. It gets a0 and a1 from GPU0 and GPU1, and with its a2 it reconstructs the full tensor.

All 3 GPUs get the full tensors reconstructed and a forward happens.

As soon as the calculation is done, the data that is no longer needed gets dropped - it's only used during the calculation. The reconstruction is done efficiently via a pre-fetch.

And the whole process is repeated for layer Lb, then Lc forward-wise, and then backward Lc -> Lb -> La.

To me this sounds like an efficient group backpacking weight distribution strategy:

1. person A carries the tent
2. person B carries the stove
3. person C carries the axe

Now each night they all share what they have with others and get from others what they don't have, and in the morning they pack up their allocated type of gear and continue on their way. This is Sharded DDP / ZeRO DP.

Compare this strategy to the simple one where each person has to carry their own tent, stove and axe, which would be far more inefficient. This is DataParallel (DP and DDP) in PyTorch.

While reading the literature on this topic you may encounter the following synonyms: Sharded, Partitioned.

If you pay close attention the way ZeRO partitions the model's weights - it looks very similar to tensor parallelism which will be discussed later. This is because it partitions/shards each layer's weights, unlike vertical model parallelism which is discussed next.

Implementations of ZeRO-DP stages 1+2+3:

- DeepSpeed

- PyTorch (originally it was implemented in FairScale and later it was upstreamed into the PyTorch core)
- torchtitan

DeepSpeed ZeRO Integration:

- HF Trainer integration
- Accelerate
- PyTorch Lightning
- Determined.AI

FSDP Integration:

- HF Trainer integration
- Accelerate
- PyTorch Lightning
- torchtitan

Important papers:

DeepSpeed and ZeRO in general:

- ZeRO: Memory Optimizations Toward Training Trillion Parameter Models
- ZeRO-Offload: Democratizing Billion-Scale Model Training
- ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning
- ZeRO++: Extremely Efficient Collective Communication for Giant Model Training
- DeepSpeed Ulysses: System Optimizations for Enabling Training of Extreme Long Sequence Transformer Models
- AMSP: Reducing Communication Overhead of ZeRO for Efficient LLM Training

PyTorch:

- PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel

Main DeepSpeed ZeRO Resources:

- Project's github
- Usage docs
- API docs
- Blog posts

Overcoming the huge global batch size issue

If you use, say, 1024 accelerators, you'll have tiny shards per accelerator and a ton of free memory for micro-batch-size (MBS), let's say you can fit MBS=32 - you end up with GBS=32k - most likely not what you want.

So you either need to deploy Tensor Parallelism which is non-trivial to implement, or often it's much simpler to deploy Sequence Parallelism. I'm yet to try it in action, but so far what I gathered is for:

- DeepSpeed ZeRO use DeepSpeed-Ulysses
- Arctic Long Sequence Training
- FSDP use Paged Ring Attention (paper)

Please note that most likely it won't be as efficient as Tensor Parallelism - but I don't yet know of the actual additional overhead.

ZeRO with multiple replicas

By default ZeRO uses all GPUs to create a single model replica - that's the model is spread out across all gpus. Which leads to various limitations such as:

1. the global batch size is inflexible - it's always a function of `total_gpus*micro_batch_size` - which on large

clusters could lead to a huge global batch size which might be detrimental for efficient convergence. Granted one could use a tiny micro batch size to keep the global batch size in check, but this leads to smaller matrices on each GPU which results in less efficient compute
2. the much faster intra-node networking is not being benefited from since the slower inter-node network defines the overall speed of communications.

ZeRO++ solves the 2nd limitation by introducing Hierarchical Weight Partition for ZeRO (hpZ). In this approach instead of spreading whole model weights across all the gpus, each model replica is restricted to a single node. This increases the memory usage by the total number of nodes, but now the 2x `all_gather` calls to gather the sharded components are performed over a much faster intra-node connection. Only the `reduce_scatter` to aggregate and redistribute gradients is performed over the slower inter-node network.

The first limitation doesn't exactly get fixed since the overall global batch size remains the same, but since each replica is more efficient and because the additional memory pressure is likely to limit the possible micro batch size on each gpu, this overall should improve the throughput of the system.

PyTorch FSDP has this feature implemented in shardingStrategy.HYBRID_SHARD

Papers:

- ZeRO++: Extremely Efficient Collective Communication for Giant Model Training
- PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel

ZeRO variations

Published papers that propose modifications to the ZeRO protocol:

- MiCS: Near-linear Scaling for Training Gigantic Model on Public Cloud (2022)
- AMSP: Super-Scaling LLM Training via Advanced Model States Partitioning (2023)

Pipeline Parallelism methods

Naive Model Parallelism (Vertical)

Naive Model Parallelism (MP) is where one spreads groups of model layers across multiple GPUs. The mechanism is relatively simple - switch the desired layers `.to()` the desired devices and now whenever the data goes in and out those layers switch the data to the same device as the layer and leave the rest unmodified.

We refer to it as Vertical MP, because if you remember how most models are drawn, we slice the layers vertically. For example, if the following diagram shows an 8-layer model:

```
==================  ==================
|  0 | 1 | 2 | 3  |  |  4 | 5 | 6 | 7  |
==================  ==================
        gpu0                gpu1
```

we just sliced it in 2 vertically, placing layers 0-3 onto GPU0 and 4-7 to GPU1.

Now while data travels from layer 0 to 1, 1 to 2 and 2 to 3 this is just the normal model. But when data needs to pass from layer 3 to layer 4 it needs to travel from GPU0 to GPU1 which introduces a communication overhead. If the participating GPUs are on the same compute node (e.g. same physical machine) this copying is pretty fast, but if the GPUs are located on different compute nodes (e.g. multiple machines) the communication overhead could be significantly larger.

Then layers 4 to 5 to 6 to 7 are as a normal model would have and when the 7th layer completes we often need to send the data back to layer 0 where the labels are (or alternatively send the labels to the last layer). Now the loss can be computed

and the optimizer can do its work.

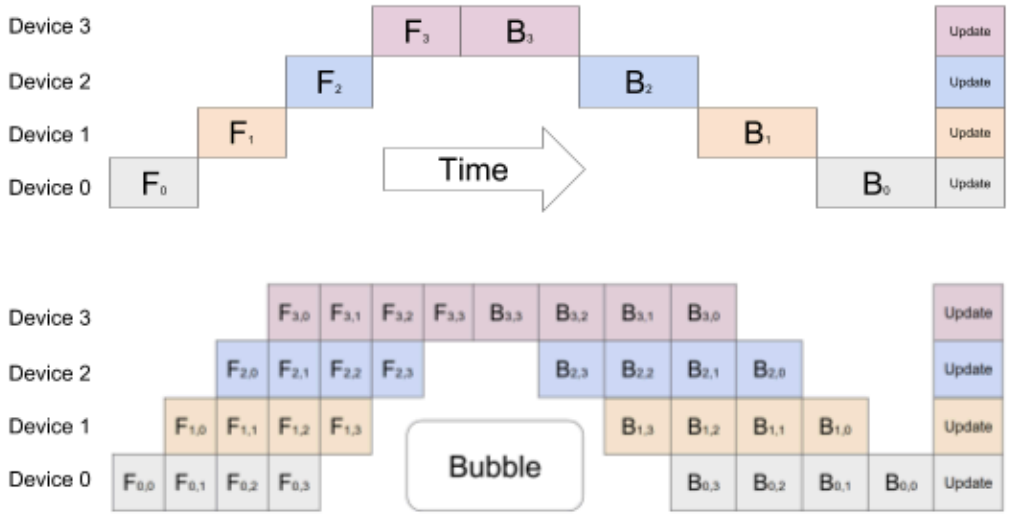
Problems:

- the main deficiency and why this one is called "naive" MP, is that all but one GPU is idle at any given moment. So if 4 GPUs are used, it's almost identical to quadrupling the amount of memory of a single GPU, and ignoring the rest of the hardware. Plus there is the overhead of copying the data between devices. So 4x 6GB cards provide roughly the same aggregate capacity for partitioned layer state as 1x 24GB card, but the single card will complete the training faster because it avoids inter-device transfers. Naive MP is mainly a capacity technique: it can make a workload fit when its layer-associated training state exceeds one GPU's memory, provided each partition and its activations fit on the assigned GPU.
- shared embeddings may need to get copied back and forth between GPUs.

Pipeline Parallelism

Pipeline Parallelism (PP) is almost identical to a naive MP, but it solves the GPU idling problem, by chunking the incoming batch into micro-batches and artificially creating a pipeline, which allows different GPUs to concurrently participate in the computation process.

The following illustration from the GPipe paper shows the naive MP on the top, and PP on the bottom:



Top: The naive model parallelism strategy leads to severe underutilization due to the sequential nature of the network. Only one accelerator is active at a time. Bottom: GPipe divides the input mini-batch into smaller micro-batches, enabling different accelerators to work on separate micro-batches at the same time.

It's easy to see from the bottom diagram how PP has less dead zones, where GPUs are idle. The idle parts are referred to as the "bubble".

Both parts of the diagram show a parallelism that is of degree 4. That is 4 GPUs are participating in the pipeline. So there is the forward path of 4 pipe stages F0, F1, F2 and F3 and then the return reverse order backward path of B3, B2, B1 and B0.

PP introduces a new hyper-parameter to tune and it's `chunks` which defines how many chunks of data are sent in a sequence through the same pipe stage. For example, in the bottom diagram you can see that `chunks=4`. GPU0 performs the same forward path on chunk 0, 1, 2 and 3 (F0,0, F0,1, F0,2, F0,3) and then it waits for other GPUs to do their work and only

when their work is starting to be complete, GPU0 starts to work again doing the backward path for chunks 3, 2, 1 and 0 (B0,3, B0,2, B0,1, B0,0).

Note that conceptually this is the same concept as gradient accumulation steps (GAS). PyTorch uses `chunks`, whereas DeepSpeed refers to the same hyper-parameter as GAS.

Because of the chunks, PP introduces the concept of micro-batches (MBS). DP splits the global data batch size into mini-batches, so if you have a DP degree of 4, a global batch size of 1024 gets split up into 4 mini-batches of 256 each (1024/4). And if the number of `chunks` (or GAS) is 32 we end up with a micro-batch size of 8 (256/32). Each Pipeline stage works with a single micro-batch at a time.

To calculate the global batch size of the DP + PP setup we then do: `mbs*chunks*dp_degree` (`8*32*4=1024`).

Let's go back to the diagram.

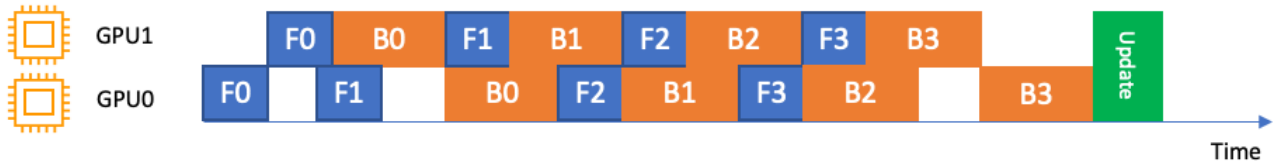
With `chunks=1` you end up with the naive MP, which is very inefficient. With a very large `chunks` value you end up with tiny micro-batch sizes which could be not every efficient either. So one has to experiment to find the value that leads to the highest efficient utilization of the gpus.

While the diagram shows that there is a bubble of "dead" time that can't be parallelized because the last `forward` stage has to wait for `backward` to complete the pipeline, the purpose of finding the best value for `chunks` is to enable a high concurrent GPU utilization across all participating GPUs which translates to minimizing the size of the bubble.

The choice of the schedule is critical to the efficient performance, with the most common schedules being in the order of invention:

- sequential Gpipe: Efficient training of giant neural networks using pipeline parallelism
- interleaved 1F1B Pipedream: Fast and efficient pipeline parallel dnn training
- looped, depth-first Efficient large-scale language model training on gpu clusters using Megatron-LM
- breadth-first Breadth-First Pipeline Parallelism
- Llama 3 training used a combination of depth and breadth first for best performance and also allowed them to progressively modify the global batch size as the training progressed, which is typically very difficult to accomplish with PP. See The Llama 3 Herd of Models section 3.3.2 Parallelism for Model Scaling.

Here is for example an interleaved pipeline:



Here the bubble (idle time) is further minimized by prioritizing backward passes.

It's used by DeepSpeed and SageMaker to name a few.

DeepSeek v3 introduced an even more efficient PP via DualPipe that reduces the bubble size and succeeds at a better compute/comms overlap. See section 3.2.1 of the paper for the specific details.



(source)

PP practical caveats

Regardless of which PP stack you use, several constraints keep showing up:

- have to modify the model quite heavily, because the normal module graph has to be cut into pipeline stages - some APIs still want something close to an `nn.Sequential` of stages - which may require changes to the design of the model.
- stage boundaries are restricted in what they can pass. If you had a bunch of python variables flowing through the first stage of a single-process forward, you will have to find a way around it. Typical interfaces expect either a single Tensor or a tuple of Tensors as stage inputs and outputs, and those tensors must have a batch size as the very first dimension, since the pipeline is going to chunk the mini-batch into micro-batches.
- conditional control flow at the level of pipe stages is not possible - e.g., Encoder-Decoder models like T5 require special workarounds to handle a conditional encoder stage.
- have to arrange each layer so that the output of one stage becomes an input to the next.
- The first stage contains a heavy embedding which can be quite huge if the vocabulary is large - and this may require a custom splicing so that the first stage will contain less transformer blocks than other stages.

Modern stacks ease some of the boilerplate, but they do not remove these structural constraints.

Where to get PP today

The practical choices are production training stacks with their own PP, and PyTorch's composable API:

- Megatron-LM - battle-tested PP (interleaved 1F1B and related schedules); model-specific, no separate public Pipe-style API
- DeepSpeed - PP as part of the DeepSpeed stack (often combined with ZeRO)
- torch.distributed.pipelining - PyTorch's current PP toolkit (migrated from PiPPy); still **alpha** / API may change. Splits general models, ships schedules including GPipe, 1F1B, interleaved 1F1B, looped BFS, and DualPipeV (DeepSeek-style), and composes with DP/FSDP/TP - see torchtitan for a 3D-parallel Llama example
- torchtitan / nanotron - full training stacks that use modern PP
- DeepSeek DualPipe - the schedule from DeepSeek-V3 (also appears as `SchedulePipeV` in `torch.distributed.pipelining`)
- SageMaker - proprietary, AWS-only

Related reading

- Pipeline-Parallelism: Distributed Training via Model Partitioning

Tensor Parallelism

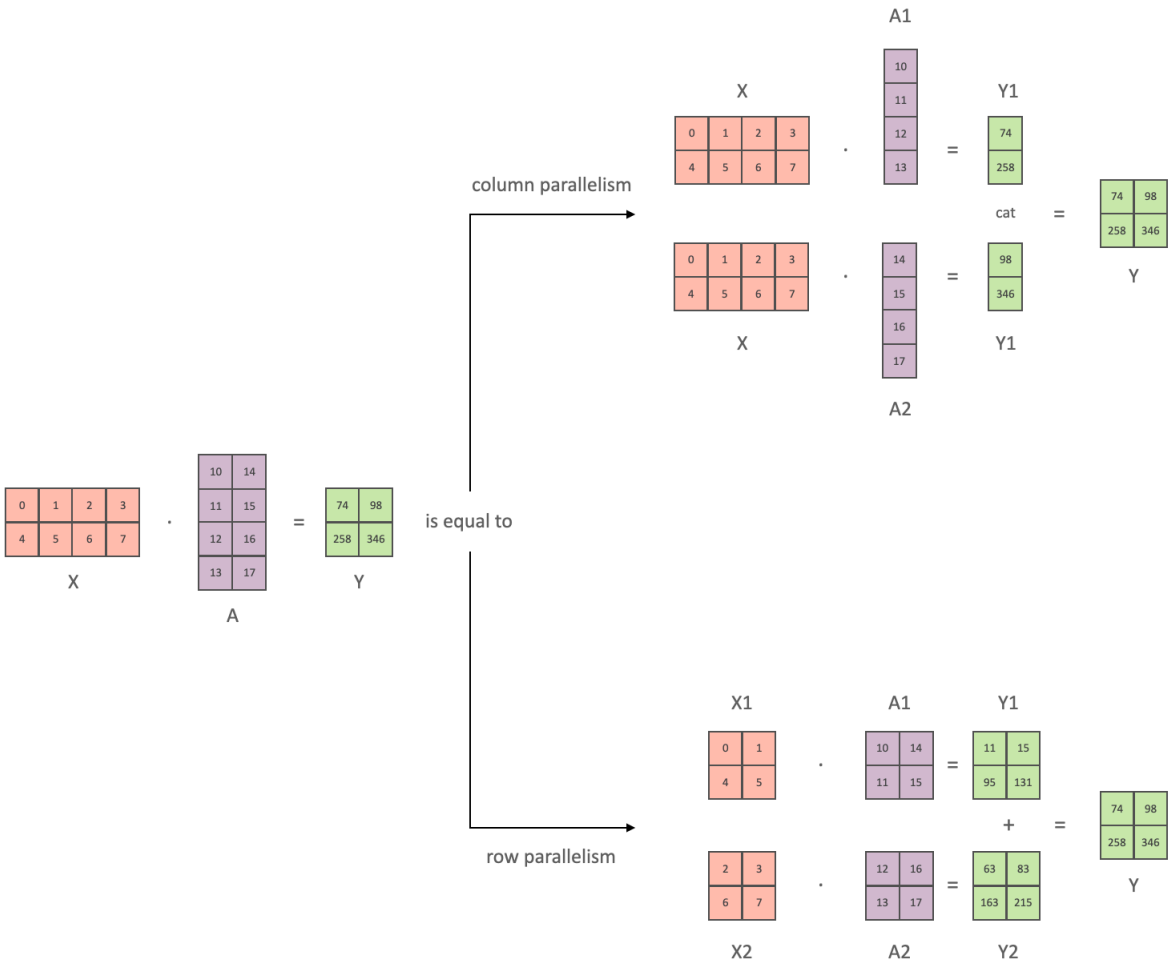
In Tensor Parallelism each GPU processes only a slice of a tensor and only aggregates the full tensor for operations that require the whole thing.

In this section we use concepts and diagrams from the Megatron-LM paper: Efficient Large-Scale Language Model Training on GPU Clusters.

The main building block of any transformer is a fully connected `nn.Linear` followed by a nonlinear activation `GeLU`.

Following the Megatron's paper notation, we can write the dot-product part of it as `Y = GeLU(XA)`, where `X` and `Y` are the input and output vectors, and `A` is the weight matrix.

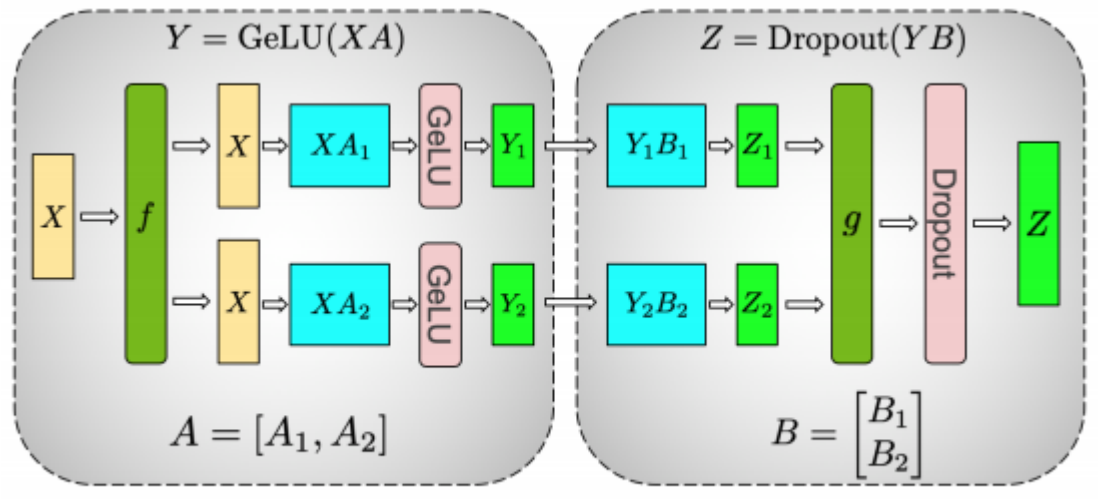
If we look at the computation in matrix form, it's easy to see how the matrix multiplication can be split between multiple GPUs:



If we split the weight matrix `A` column-wise across `N` GPUs and perform matrix multiplications `XA_1` through `XA_n` in parallel, then we will end up with `N` output vectors `Y_1, Y_2, ..., Y_n` which can be fed into `GeLU` independently:

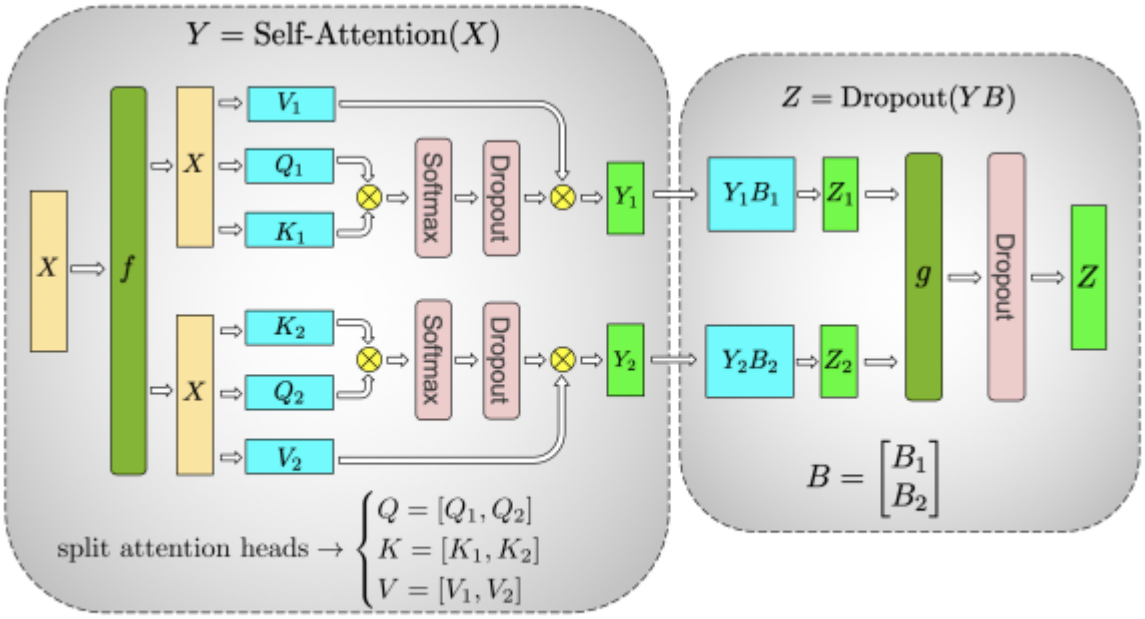
$$[Y_1, Y_2] = [\text{GeLU}(XA_1), \text{GeLU}(XA_2)]$$

Using this principle, we can update an MLP of arbitrary depth, without the need for any synchronization between GPUs until the very end, where we need to reconstruct the output vector from shards. The Megatron-LM paper authors provide a helpful illustration for that:



(a) MLP

Parallelizing the multi-headed attention layers is even simpler, since they are already inherently parallel, due to having multiple independent heads!



(b) Self-Attention

Important: TP requires very fast network, and therefore since typically intra-node networks are much faster than inter-node networks it's not advisable to do TP across nodes. Practically, if a node has 4 GPUs, the highest TP degree is therefore 4. If you need a TP degree of 8, you need to use nodes that have at least 8 GPUs.

TP can be combined with other parallelization methods.

Alternative names:

- DeepSpeed calls it tensor slicing

Implementations:

- Megatron-LM has an internal implementation, as it's very model-specific
- PyTorch

- SageMaker - this is a proprietary solution that can only be used on AWS.
- OSLO has the tensor parallelism implementation based on the Transformers.
- nanotron
- torchtitan

Async TP

One of the deficiencies of TP is that it's difficult to overlap its comms with compute. PyTorch is proposing to overcome this with Async-TP which decomposes the dependent sequence of all-gather + matmul into series of cudaMemcpyAsync calls and smaller partial matmuls - and it does it automatically for you using `torch.compile`!

- Megatron-LM has it implemented as well via `--tp-comm-overlap`.

Related reading

- Tensor Parallelism and Sequence Parallelism: Detailed Analysis

Sequence Parallelism

ML tasks, such as DNA sequencing, may require training with very long sequence lengths (e.g. 256K), and even normal LLMs could be trained on sequences of 10k and longer.

Self-Attention, which is a key component of Transformers, has quadratic compute complexity with respect to sequence length for standard dense attention. Kernels that materialize the full sequence-by-sequence attention matrix also require quadratic temporary memory. Memory-efficient kernels such as FlashAttention avoid storing that matrix and keep attention memory linear in sequence length without removing the quadratic compute.

At sufficiently long sequence lengths, even a batch size of 1 may not fit on a single GPU, so sequence parallelism partitions work along the sequence dimension. This increases the supported sequence length, but does not make it unlimited: the practical limit still depends on aggregate accelerator memory, compute time, communication overhead, and kernel constraints.

As this type of parallelism is orthogonal to the other parallelization types described in this document, it can be combined with any of them, leading to 4D, ZeRO-DP+SP and other combinations.

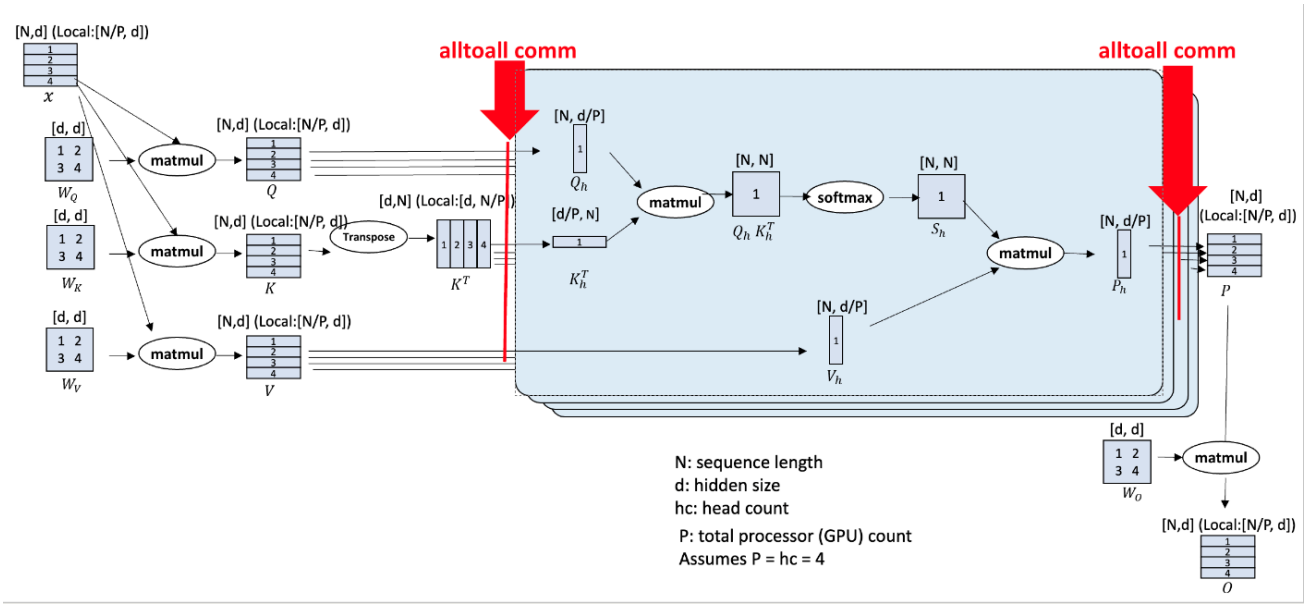
DeepSpeed-Ulysses SP

Paper: DeepSpeed Ulysses: System Optimizations for Enabling Training of Extreme Long Sequence Transformer Models

In this implementation 2 elements are sharded:

1. The multiple-head attention weights are split across the participating GPUs so that each GPU has a few sub-heads only. This is done when the model is created/loaded. This is somewhat similar to Tensor Parallelism.
2. During training each input sequence is partitioned into chunks and each chunk is sent to one of the GPUs, which reminds us of ZeRO-3 sharding, except instead of weights the inputs are sharded.

During compute each sequence chunk is projected onto QKV and then gathered to the full sequence QKV on each device, computed on each device only for the subheads it owns and then gathered again into the full attention output for the MLP block.



source

On the diagram:

1. Input sequences N are partitioned across P available devices.
2. Each local N/P partition of the input sequence is projected into queries (Q), keys (K) and values (V) embeddings.
3. Next, local QKV embeddings are gathered into global QKV through highly optimized all-to-all collectives between participating compute devices.
4. Then the attention computation per head is performed:

$$Output\ context = Softmax\ (\frac{QK^T}{\sqrt{d}})V$$

5. At the end another all-to-all collective transforms output context tensor of attention computation to sequence (N/P) parallel for subsequent operators (MLP MatMul, layer norm, etc.) in the remaining modules of transformer layer block.

Example: Let's consider seqlen=8K, num_heads=128 and a single node of num_gpus=8

1. each GPU gets a 1K-long chunk of the original sequence (`8K/8`)
2. each GPU gets assigned 16 sub-heads (`128/8`)
3. a. on gpu0 before `forward` the original sequence is gathered back into 8K tokens b. the attention computation is done on the first 16 sub-heads

The same logic is performed on the remaining 7 GPUs, each computing 8k attention over its 16 sub-heads

You can read the specifics of the very efficient comms here.

DeepSpeed-Ulysses keeps communication volume consistent by increasing GPUs proportional to message size or sequence length.

Arctic Long Sequence Training

Arctic Long Sequence Training ports DeepSpeed-Ulysses to HuggingFace Transformers, while updating it to work with

modern attention head mechanisms and extends it further to enable a much longer sequence length support (or batch size) by tiling compute and offloading the activation checkpoints. The integration guide is here.

- paper: https://arxiv.org/abs/2506.13996
- implementation and integration: ArcticTraining and Axolotl

Colossal-AI's SP

Paper: Sequence parallelism: Long sequence training from system perspective

Colossal-AI's SP implementation uses ring self-attention, a ring-like communication collective in which query projections are local whereas key and values projections are transmitted in a ring-style to compute global attention, resulting in communication complexity linear in message size, M.

Megatron-LM's SP

Paper: Reducing Activation Recomputation in Large Transformer Models

Megatron-LM's SP is tightly integrated with its TP. Megatron-LM partitions sequence along sequence dimensions and applies allgather and reduce scatter collective to aggregate QKV projections for attention computation. Its communication volume increases linearly with message size (M) regardless of number of compute devices.

Ring Attention with Blockwise Transformers

Paper: Ring Attention with Blockwise Transformers for Near-Infinite Context

Also known as Ring Self-Attention (RSA), the name coined in the Colossal-AI's SP paper that proposed the same ring-style passing of keys and values while the queries stay local. Both names are in common use, so DistFlashAttn's comparison against Ring Self-Attention refers to this technique.

1. Tensors are sharded along the sequence dimension throughout: (`seq_len // N, d_model`)-shaped
2. In the attention layers, every GPU starts by computing the part of the attention scores they are able to w/ their available shards.
3. Simultaneously, the keys and values from other sequence chunks are communicated around.
4. Once the keys/values from another chunk are available, each GPU continues on with their attention computation using the key/value tensors from this new segment of the sequence
5. Continue until attention computation is complete.

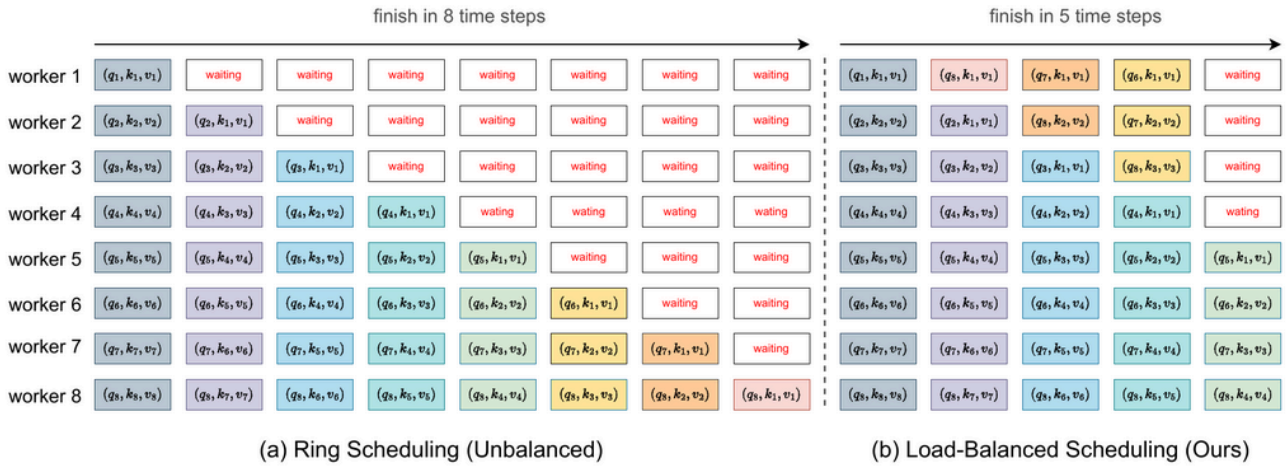
SP Implementations:

- Megatron-LM
- DeepSpeed
- Colossal-AI
- torchtitan

PyTorch is also working on this feature and calling it Context Parallel (CP).

DistFlashAttn

DISTFLASHATTN: Distributed Memory-efficient Attention for Long-context LLMs Training is reported to be many times faster than Ring Self-Attention, because it load balances the KVQ per token computation between the workers while performing Sequence Parallelism.



Related reading

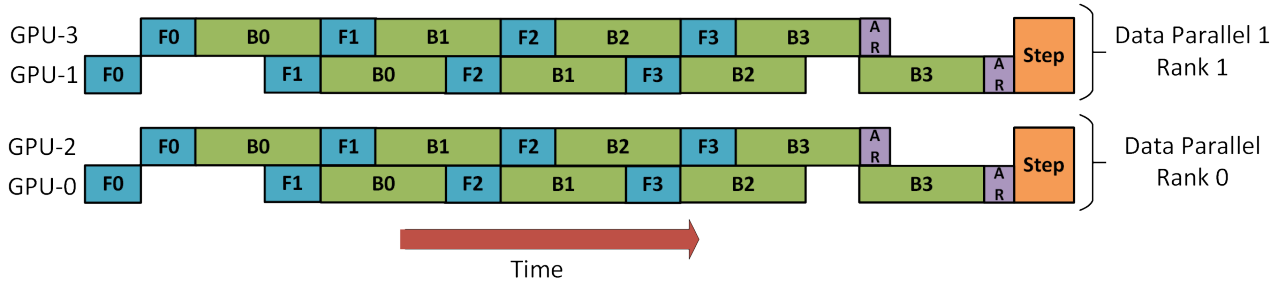
- Tensor Parallelism and Sequence Parallelism: Detailed Analysis

TP+SP

TP can be combined with SP in the same process group to minimize communication costs as explained in Reducing Activation Recomputation in Large Transformer Models. For example in LLMs, TP is used for embedding, attention and linear layers and when dropout and layer norm are reached SP is used instead.

DP+PP

The following diagram from the DeepSpeed pipeline tutorial demonstrates how one combines DP with PP.



Here it's important to see how DP rank 0 doesn't see GPU2 and DP rank 1 doesn't see GPU3. To DP there is just GPUs 0 and 1 where it feeds data as if there were just 2 GPUs. GPU0 "secretly" offloads some of its load to GPU2 using PP. And GPU1 does the same by enlisting GPU3 to its aid.

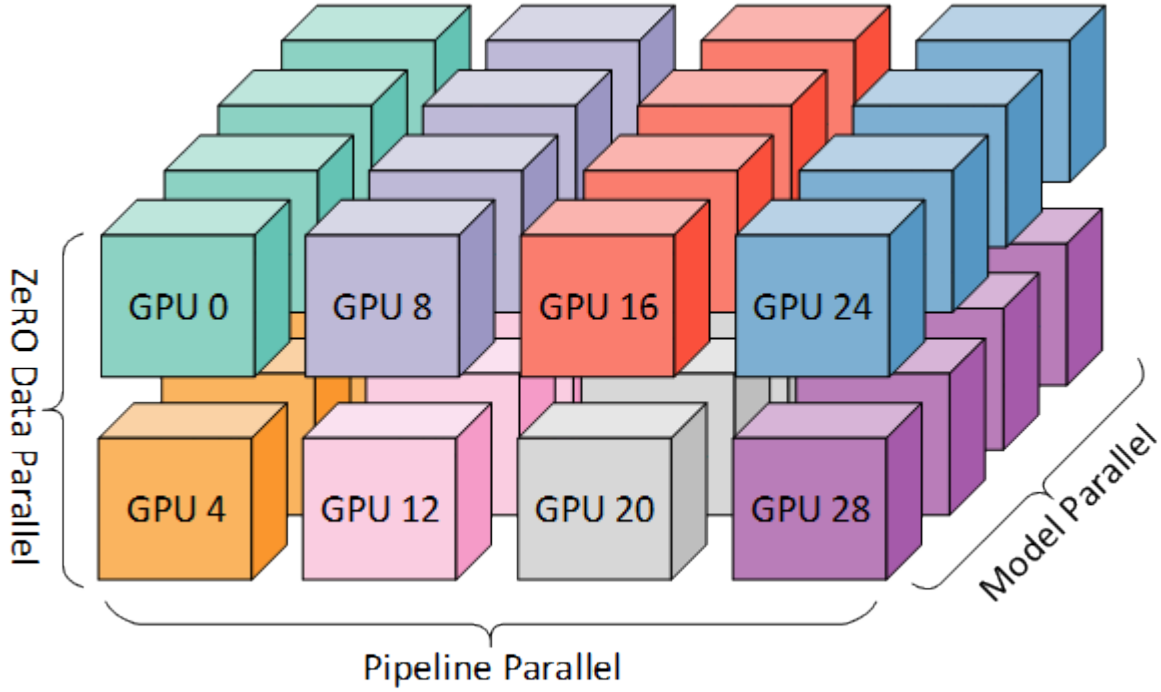
Since each dimension requires at least 2 GPUs, here you'd need at least 4 GPUs.

Implementations:

- DeepSpeed
- Megatron-LM
- SageMaker
- OSLO
- nanotron
- torchtitan

DP+PP+TP

To get an even more efficient training a 3D parallelism is used where PP is combined with TP and DP. This can be seen in the following diagram.



This diagram is from a blog post 3D parallelism: Scaling to trillion-parameter models, which is a good read as well.

Since each dimension requires at least 2 GPUs, here you'd need at least 8 GPUs.

Implementations:

- DeepSpeed - DeepSpeed also includes an even more efficient DP, which they call ZeRO-DP.
- Megatron-LM
- SageMaker
- OSLO
- nanotron
- torchtitan

ZeRO DP+PP+TP

One of the main features of DeepSpeed is ZeRO, which is a super-scalable extension of DP. It has already been discussed in ZeRO Data Parallelism. Normally it's a standalone feature that doesn't require PP or TP. But it can be combined with PP and TP.

When ZeRO-DP is combined with PP (and optionally TP) it typically enables only ZeRO stage 1 (optimizer sharding).

While it's theoretically possible to use ZeRO stage 2 (gradient sharding) with Pipeline Parallelism, it will have bad performance impacts. There would need to be an additional reduce-scatter collective for every micro-batch to aggregate the gradients before sharding, which adds a potentially significant communication overhead. By nature of Pipeline Parallelism, small micro-batches are used and instead the focus is on trying to balance arithmetic intensity (micro-batch size) with minimizing the Pipeline bubble (number of micro-batches). Therefore those communication costs are going to hurt.

In addition, there are already fewer layers than normal due to PP and so the memory savings won't be huge. PP already

reduces gradient size by `1/PP`, and so gradient sharding savings on top of that are less significant than pure DP.

ZeRO stage 3 is not a good choice either for the same reason - more inter-node communications required.

And since we have ZeRO, the other benefit is ZeRO-Offload. Since this is stage 1 optimizer states can be offloaded to CPU.

Implementations:

- Megatron-DeepSpeed and Megatron-DeepSpeed from BigScience, which is the fork of the former repo.
- OSLO
- torchtitan

Important papers:

- Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model

Expert Parallelism

When Mixture-Of-Experts (MoE) is used (in particular during inference) one could give each expert its own accelerator (or a few if one isn't enough), which is referred to as Expert Parallelism (EP). This adds another dimension for parallelization and can significantly speed things up for large batches that are likely to hit all of the experts. Instead of communicating model weights, in EP tokens are being communicated instead. EP leads to a more efficient compute as matrix multiplication then deal with bigger inputs.

For detailed explanations please see:

- DeepSpeed-MoE: Advancing Mixture-of-Experts Inference and Training to Power Next-Generation AI Scale
- Mixture of Experts Explained

Other parallelism approaches

FlexFlow

FlexFlow also solves the parallelization problem in a slightly different approach.

Paper: "Beyond Data and Model Parallelism for Deep Neural Networks" by Zhihao Jia, Matei Zaharia, Alex Aiken

It performs a sort of 4D Parallelism over Sample-Operator-Attribute-Parameter.

1. Sample = Data Parallelism (sample-wise parallel)
2. Operator = Parallelize a single operation into several sub-operations
3. Attribute = Data Parallelism (length-wise parallel)
4. Parameter = Model Parallelism (regardless of dimension - horizontal or vertical)

Examples:

- Sample

Let's take 10 batches of sequence length 512. If we parallelize them by sample dimension into 2 devices, we get 10 x 512 which becomes be 5 x 2 x 512.

- Operator

Layer normalization requires the mean and variance, so standard deviation and mean aren't independent operations. For an illustrative operator decomposition across 2 devices (cuda:0, cuda:1), first copy the input data to both devices, then let cuda:0 reduce `sum(x)` while cuda:1 reduces `sum(x*x)`. The results can be combined as `mean = E[x]` and `variance = E[x*x] - E[x]**2`. This direct moment formula can lose numerical precision, so production implementations typically use a stable reduction algorithm instead. See the PyTorch `torch.std_mean` documentation.

- Attribute

We have 10 batches of 512 length. If we parallelize them by attribute dimension into 2 devices, 10 x 512 will be 10 x 2 x 256.

- Parameter

It is similar with tensor model parallelism or naive layer-wise model parallelism.

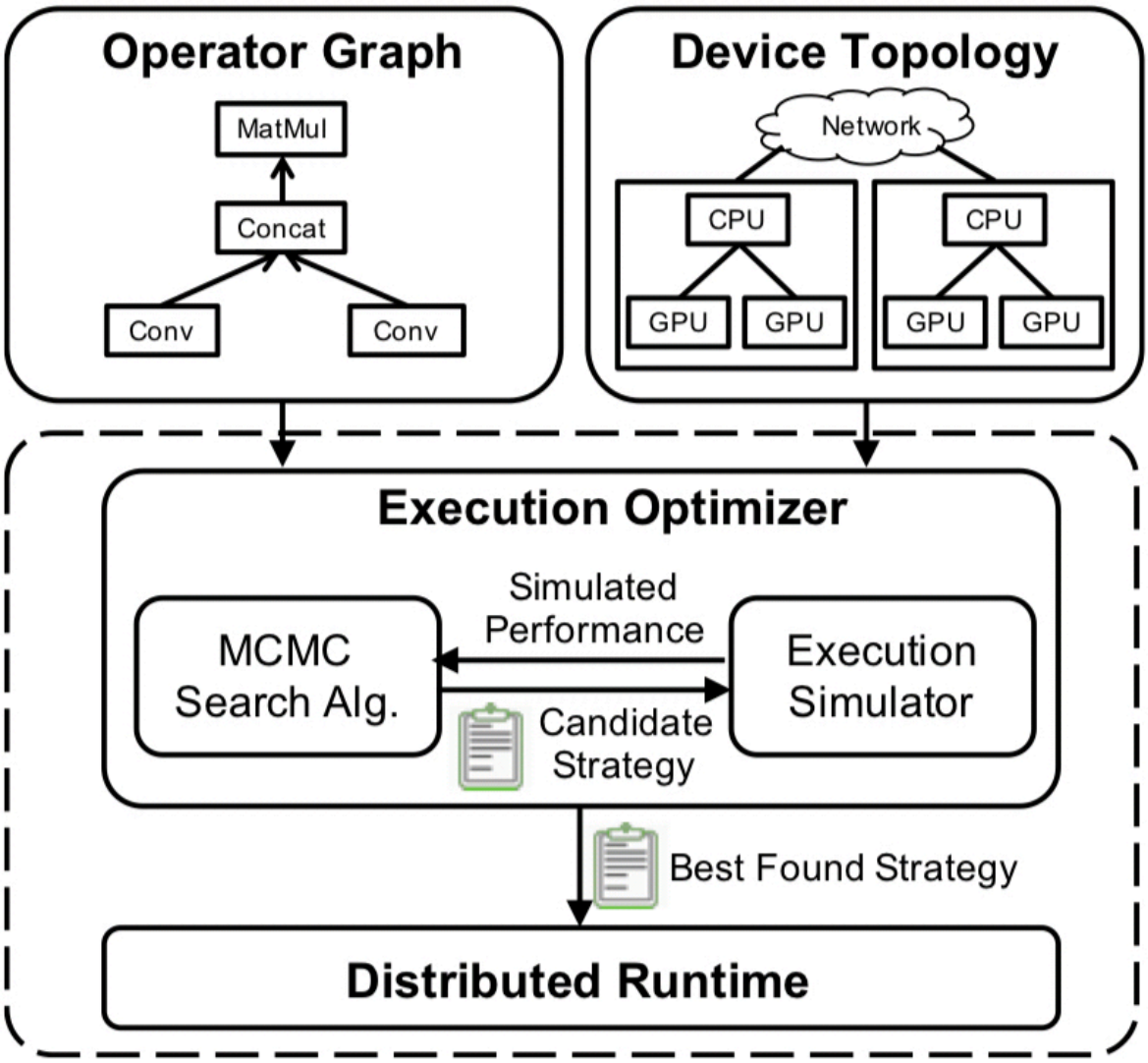


Figure 1. FlexFlow overview.

The significance of this framework is that it takes resources like (1) GPU/TPU/CPU vs. (2) RAM/DRAM vs. (3) fast-intra-connect/slow-inter-connect and it automatically optimizes all these algorithmically deciding which parallelisation to use where.

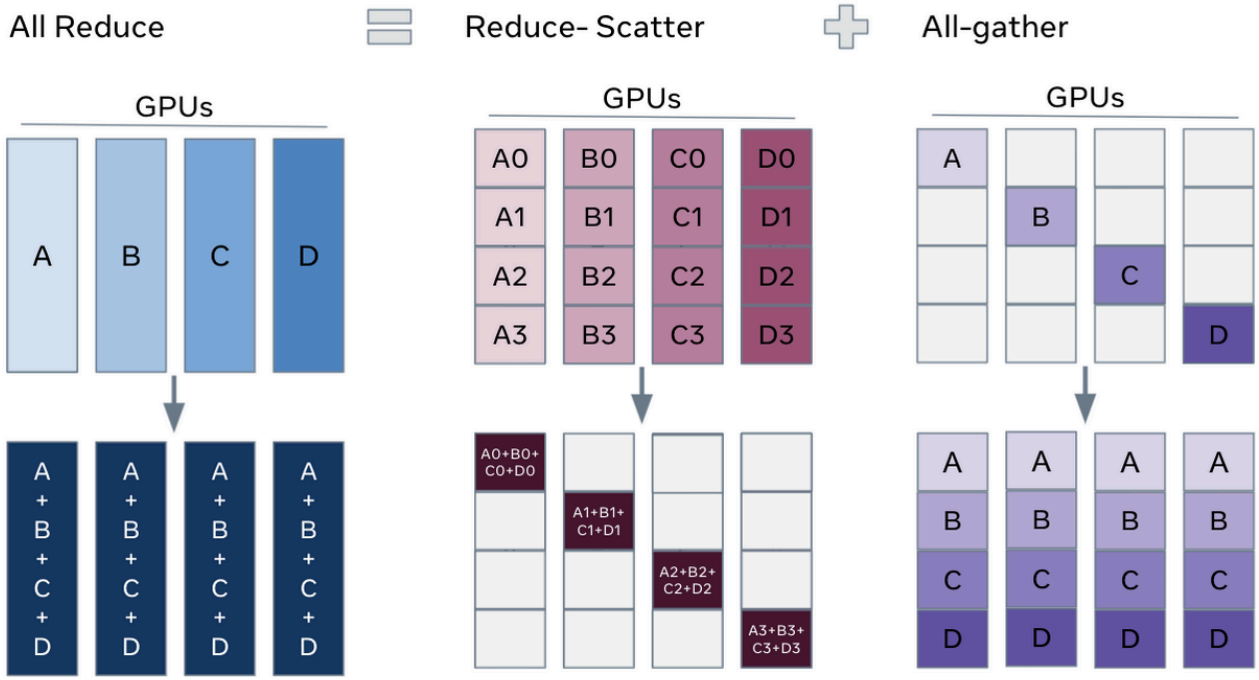
One very important aspect is that FlexFlow is designed for optimizing DNN parallelizations for models with static and fixed workloads, since models with dynamic behavior may prefer different parallelization strategies across iterations.

So the promise is very attractive - it runs a 30min simulation on the cluster of choice and it comes up with the best strategy to utilise this specific environment. If you add/remove/replace any parts it'll run and re-optimize the plan for that. And then you can train. A different setup will have its own custom optimization.

Parallelism network collectives

As intra- and inter-node speeds have an order of magnitude difference, it's crucial to choose different parallelization techniques for intra- and inter-node crossing. e.g. TP should stay inside the fastest, lowest-latency domain because of its massive synchronization requirements - traditionally that means within the node, but on rack-scale systems like NVL72 that domain spans many nodes (called supernode), so the real rule is to keep TP inside the scale-up fabric and to cross into the slower scale-out network only when measurement says you can afford it or capacity leaves no choice. Moreover, some accelerators, like the recent AMD MI3** series have a very slow gpu-to-gpu connectivity which again impacts how parallelism is done for the best performance.

Here is a useful tidbit: the all-reduce collective can be decomposed into two separate phases: reduce-scatter and all-gather.



(source)

Here is the breakdown of which collectives are used for which parallelization strategies:

- DDP: 1x all-reduce for the gradients - ideally overlapping with compute - total volume: 2x model params comms
- ZeRO-DP ZeRO-1/ZeRO-2: 1x reduce-scatter for gradients plus 1x all-gather to collect all the updated parameters- total volume: 2x model params comms
- ZeRO-DP ZeRO-3: 2x all-gather for weights (before `forward` + before `backward`) plus 1x reduce-scatter for gradients - total volume: 3x model params comms (1.5x more than DDP and ZeRO-1/ZeRO-2)
- TP: 2x all-gather and 2x reduce-scatter
- PP: 2x send + 2x recv - overlapping with compute in the steady phase
- SP: depends on implementation: with hidden size h, sequence length of N, and parallelism degree of P
    - Megatron-LM: 2x all-gather and 2x reduce-scatter with volume `4*N*h` per transformers layer section 3.2 in paper
    - DeepSpeed Ulysses: 2x all-to-all with volume `4*N*h/P` per transformers layer section 3.2 in paper

It's possible that you will find different implementations that may use different communication patterns.

Inter-node speed requirements to use ZeRO

The ZeRO scalability protocol, be it DeepSpeed ZeRO or PyTorch FSDP, requires a lot more inter-node traffic than TP+PP+DP solutions, and sometimes it can't take advantage of the faster intra-node connectivity, and therefore if your inter-node network is slow your expensive GPUs might be massively bottlenecked by the comms.

The ZeRO protocol partially overlaps comms with compute, so ideally you want to get close to `comms_time <= compute_time`. The overlap is not perfect, so there will be always some network bottleneck, but we want to make sure that `comms_time` is not much larger than `compute_time`.

In ZeRO-3, we have `all_gather` on weights in `forward`, then `all_gather` on weights in `backward`, last is `reduce_scatter` on gradients in backward. In total there are 3 global collective calls each sending a model size multiplied by how many bytes per parameter are used. e.g. a 10B param model in bf16 under ZeRO-3 will need to send `10*2*3` = 60GB of data.

In comparison DistributedDataParallel (DDP) uses a single `all_reduce` call, but which requires 2x data transmission, and so a 10B param model in bf16 under DDP will need to send `10*2*2` = 40GB of data.

ZeRO-1 which only shards the optimiser states, like DDP, will too need to transmit 40GB of data (one `all_gather` and one `reduce_scatter`.)

Here is how to calculate time in seconds for communication and compute:

- `comms_time = comms_multiplier * n_bytes * model_size_in_B / inter-node-throughput_in_GBps`
- `compute_time = n_passes * n_bytes * model_size_in_B * seqlen * global_batch_size / (total_gpus * 1e3 * tflops_wo_comms)`

The compute time formula is a rough estimate which works for any Transformer-block based model. It ignores any small computations and includes only the massive `matmul`s.

The comms time formula is also a first-order estimate. `comms_multiplier` is the sum of the busbw correction factors of the collectives performed in a single step - `2(n-1)/n` for `all_reduce` and `(n-1)/n` each for `all_gather` and `reduce_scatter`, where `n` is the number of ranks - which is where DDP's 2 and ZeRO-3's 3 come from. It drops the `(n-1)/n` part, which is a good approximation for a large number of ranks (0.998 at the 512 ranks used below), but too optimistic for a small one (0.875 at 8 ranks). And the throughput you plug in has to be the measured `busbw` rather than `algbw`, since `algbw` shrinks as ranks are added - though even `busbw` can vary somewhat between collectives at the same payload, because NCCL may pick a different algorithm or protocol for each.

As an experiment let's use the data points from IDEFICS-80B training.

When we trained IDEFICS-80B with a 340Gbps EFA we were getting only 90TFLOPS w/ DeepSpeed ZeRO-3 on A100s as compared to 150+TFLOPS one was getting with Megatron's TP+PP+DP. and moreover a big chunk of the model was frozen as were building a new models based on one language and one vision model. So our multiplier was less than 3. On the other hand we were using activation recomputation to save memory, so this is an additional transmission of all model weights and to top it all off since nccl wasn't supporting proper half-precision reduction we used fp32 for gradient reductions, so really our multiplier wasn't 3 but more like 4.5.

Values used for IDEFICS-80B training:

- `model_size_in_B` = 80
- `n_bytes` = 2 in case of bf16 which is 2 bytes
- `comms_multiplier` = 3 in the case of ZeRO-3/FSDP (1x reduce_scatter + 2x all_gather (fwd + bwd)) and 2 in case of ZeRO-1 (1x reduce_scatter + 1x all_gather),
- additionally, in the case of IDEFICS-80B we decided to reduce grads in fp32 to minimize NCCL accumulation loss, so we actually had `comms_multiplier*n_bytes=3*2+2=4*2` for the additional 2 bytes but since half the model was frozen only about half of gradients were sent, so we still have the multiplier of 3.
- `n_passes` = 4 with activation recomputation, or 3 w/o it. The model has to do only 1x compute per `forward` and 2x per `backward` (since the grads are calculated twice - once wrt inputs and once wrt weights). And with activation recomputation one more `forward` is done.

- `total_gpus` = `512`
- `global_batch_size` = `3584`
- `seqlen` = `1024`
- `inter-node-throughput_in_GBps` = 42.5 (340Gbps) (AWS EFA v1) `-tflops_wo_comms` is the tflops w/o the communication overhead. Not theoretical peak as that is unachievable, but perhaps ~80% in the case of A100@BF16 - so `312*0.8=~250` TFLOPS (rounded up).

We derived 340Gbps inter-node network throughput using all_reduce_bench.py with a payload of 4GiB. In the case of IDEFICS-80B we had 80 layers, so approximately each layer was 1B params large. Which means that each layer was sending 2GB of data for bf16 tensors and 4GB of data with fp32 tensors, which matches the network benchmark. If you were to have a much smaller layer size, I'd recommend adapting the benchmark to that size. For example, if your layer size was only 100M param large, then your payload would be 0.2GB for bf16 tensors. As this is an order of magnitude smaller, the network is likely to give you a lower bandwidth, and you should use that in your calculations.

footnote: if parts of your model are frozen, then there will be less data sent in syncing the gradients. in IDEFICS we had more than half of the model frozen, so when grads were reduced we only had about half the traffic.

Which gives us:

- comms = `3 * 2 * 80 / 42.5` = 11 sec
- compute = `4 * 2 * 80 * 1024 * 3584 / (512 * 1e3 * 250)` = 18 sec

If we check against our IDEFICS-80B logs, we see that each iteration took about 49 seconds.

So the good news is that the math checks out as comms + compute are in the ballpark of the measured time.

We can do another sanity check by feeding the compute formulae 90TFLOPS that we logged, in which case:

- compute = `4 * 2 * 80 * 1024 * 3584 / (512 * 1e3 * 90)` = 51 sec

and so 49 and 51 secs are pretty close. Except this tells us nothing since the logged TFLOPS were calculated using this formula, so, of course, it should match.

What I'd expect in the best case is where I have used close to theoretical peak TFLOPS in the formula and received the compute estimate to be about the same as the actual compute time measured on the system. Remember that since comms are interleaved with compute, when we measure `forward+backward` wallclock time it includes comms in it.

What's the conclusion? I'd say more investigation is needed as clearly there are additional hidden bottlenecks here. I no longer have access to this setup to investigate, so I will repeat this exercise afresh when I train another largish model and share the updated math with you. But this workout should give you a feeling for what's going on behind the scenes and how all these numbers work together.

Also this discussion didn't include into the math gradient accumulation steps (GAS). In the case of IDEFICS-80B it wasn't used. If GAS>1 the theoretical compute time doesn't change, but comms time instead of `3*2*M/GBps` would become `GAS*3*2*M/GBps`. The weights gathering via `all_gather` for `forward` and `backward` would transpire as many times as there are gradient accumulation steps. In theory for grads it'd need to happen only once, but since there is no place to store intermediary grads of the gathered weight on each GPU it'll have to be reduced GAS times as well. This is for ZeRO-2 and ZeRO-3. For ZeRO-1 GAS>1 requires no additional comms.

We also didn't discuss the `DataLoader` as a potential bottleneck here, but we tested that it was under 1 sec, i.e. a very small overhead.

Going back to comms math, we also didn't take into an account various hardware latencies, but when dealing with a large payloads they shouldn't add up a significant additional overhead.

And now you know how long it'll take to transmit that many GBs over the network of your system. For example, if the network were to be 5x slower than the one we used for IDEFICS-80B training, that is 8.5GBps (68Gbps) then:

- comms = `3 * 2 * 80 / 8.5` = 56 sec

which would definitely be a huge bottleneck compared to the faster compute.

If the network were to be 5x faster, that is 212GBps (1700Gbps) then:

- comms = `3 * 2 * 80 / 212` = 2 sec

which would be insignificant comparatively to the compute time, especially if some of it is successfully overlapped with the compute.

Also the DeepSpeed team empirically benchmarked a 176B model on 384 V100 GPUs (24 DGX-2 nodes) and found that:

1. With 100Gbps IB, we only have <20TFLOPS per GPU (bad)
2. With 200-400Gbps IB, we achieve reasonable TFLOPS around 30-40 per GPU (ok)
3. For 800Gbps IB, we reach 40+TFLOPS per GPU (excellent)

To remind the peak TFLOPS for NVIDIA V100 at fp16 is 125TFLOPS.

But be careful here - this benchmark is for V100s! Which is about 2-3x slower than A100, and 4-8x slower than H100 for half-precision. So the comms have to be at least 4-8x faster for H100 nodes to match the above table at half precision. We need more benchmarks with more recent hardware.

footnote: the 2-3x range is because the official specs claim 3x TFLOPS increase for V100->A100, and A100->H100 each, but users benchmarking the difference report at most 2.5x improvements.

They also noticed that when training at scale, the communication overhead is more pronounced with small micro-batch size per GPU. And we may not be able to increase micro-batch size since global-batch size is often fixed to achieve good model convergence rate. This is solved by the recently introduced ZeRO++.

Finally, when doing the math above you need to know the actual bandwidth you get on your setup - which changes with payload size - the larger the payload the better the bandwidth. To get this information you need to look at your `reduce_bucket_size` and `prefetch_bucket_size` settings in the DeepSpeed configuration file for reduction and prefetch correspondingly. The default is 0.5B params, which is 1GB in half-precision (0.5B x 2 bytes), or 2GB (0.5B x 4 bytes) if you use fp32 precision. So in order to measure the actual throughput you need to run an `all_reduce` benchmark with that payload and see what bandwidth gets reported. Then you can feed it to the calculations above.

Which Strategy To Use When

Here is a very rough outline at which parallelism strategy to use when. The first on each list is typically faster.

Single GPU

- Model fits onto a single GPU:
    1. Normal use
- Model doesn't fit onto a single GPU:
    1. ZeRO + Offload CPU and optionally NVMe
    2. as above plus Memory Centric Tiling (see below for details) if the largest layer can't fit into a single GPU
- Largest Layer not fitting into a single GPU:
    1. ZeRO - Enable Memory Centric Tiling (MCT). It allows you to run arbitrarily large layers by automatically splitting them and executing them sequentially. MCT reduces the number of parameters that are live on a GPU, but it does not affect the activation memory. As this need is very rare as of 2026-08 a manual override of `torch.nn.Linear` needs to be done by the user.

Single Node / Multi-GPU

- Model fits onto a single GPU:

1. DDP - Distributed DP
2. ZeRO - may or may not be faster depending on the situation and configuration used

- Model doesn't fit onto a single GPU:

  1. PP
  2. ZeRO
  3. TP

  With very fast intra-node connectivity of NVLINK or NVSwitch all three should be mostly on par, without these PP will be faster than TP or ZeRO. The degree of TP may also make a difference. Best to experiment to find the winner on your particular setup.

  TP is almost always used within a single node. That is TP size <= gpus per node.

- Largest Layer not fitting into a single GPU:

  1. If not using ZeRO - must use TP, as PP alone won't be able to fit.
  2. With ZeRO see the same entry for "Single GPU" above

Multi-Node / Multi-GPU

- If the model fits into a single node first try ZeRO with multiple replicas, because then you will be doing ZeRO over the faster intra-node connectivity, and DDP over slower inter-node
- When you have fast inter-node connectivity:

  1. ZeRO - as it requires close to no modifications to the model
  2. PP+TP+DP - less communications, but requires massive changes to the model

- when you have slow inter-node connectivity and still low on GPU memory:

  1. DP+PP+TP+ZeRO-1

Software Tune Up For The Best Performance

The faster you can make your model to train the sooner the model will finish training, which is important not only to being first to publish something, but also potentially saving a lot of money.

In general maximizing throughput is all about running many experiments and measuring the outcome and choosing the one that is superior.

In certain situations your modeling team may ask you to choose some hyper parameters that will be detrimental to throughput but overall beneficial for the overall model's success.

Glossary and concepts

- HFU: Hardware FLOPS Utilization
- MFU: Model FLOPS Utilization

MACs vs FLOP vs FLOPS vs FLOP/s

This section is here to try to disambiguate the common performance metric definitions and their relationship to each other.

MAC vs FLOP:

- 1 FLOP (FLoating point OPeration) can be one of addition, subtraction, multiplication, or division operation.
- 1 MAC (Multiply-ACCumulate) operation is a multiplication followed by an addition, that is: `a * b + c`

Thus 1 MAC = 2 FLOPs. It's also quite common for modern hardware to perform 1 MAC in a single clock cycle.

Please note that to calculate the number of MACs in relationship to FLOPs the reverse logic applies, that is MACs = 0.5 FLOPs - it's somewhat confusing since we have just said that 1 MAC = 2 FLOPs, but it checks out - observe: 100 FLOPs = 50 MACs - because there are 2 FLOPs in each MAC.

Moreover, while 1 MAC = 2 FLOPs, the reverse isn't necessarily true. That is 2 FLOPs isn't necessarily equal to 1 MAC. For example, if you did `.5*.6` 100 times it'd be 100 FLOPs, which here would equal to 100 MACs, because here only the multiply part of the MAC is executed.

FLOP vs FLOPS vs FLOP/s

- 1 FLOP (FLoating point OPeration) is any floating point addition, subtraction, multiplication, or division operation.
- 1 FLOPS (FLoating point OPeration per Second) is how many floating point operations were performed in 1 second - see FLOPS

Further you will find the following abbreviations: GFLOPS = Giga FLOPS, TFLOPS = Tera FLOPS, etc., since it's much easier to quickly grasp 150TFLOPS rather than 150000000000000FLOPS.

There is an ambiguity when FLOPS is used in writing - sometimes people use it to indicate the total quantity of operations, at other times it refers to operations per second. The latter is the most common usage and that is the definition used in this book.

In scientific writing FLOP/s is often used to clearly tell the reader that it's operations per second. Though this particular approach is hard to convert to a variable name since it still becomes `flops` when illegal characters are removed.

In some places you might also see FLOPs, which again could mean either, since it's too easy to flip lower and upper case `s`.

If the definition is ambiguous try to search for context which should help to derive what is meant:

- If it's a math equation and there is a division by time you know it's operations per second.
- If speed or performance is being discussed it usually refers to operations per second.
- If it talks about the amount of compute required to do something it refers to the total amount of operations.

TFLOPS as a performance metric

Before you start optimizing the performance of your training setup you need a metric that you can use to see whether the throughput is improving or not. You can measure seconds per iteration, or iterations per second, or some other such timing, but there is a more useful metric that measures TFLOPS.

Measuring TFLOPS is superior because without it you don't know whether you are close to the best performance that can be achieved or not. This measurement gives you an indication of how far you're from the peak performance reported by the hardware manufacturer.

In this section I will use BLOOM's training for the exemplification. We used 80GB A100 NVIDIA GPUs and we trained in mixed bf16 regime. So let's look at the A100 spec which tells us:

```
BFLOAT16 Tensor Core          312 TFLOPS
```

Therefore we now know that if we were to only run `matmul` on huge bf16 matrices of very specific dimensions without copying to and from the device we should get around 312TFLOPS max.

Practically though, due to disk IO, communications and copying data from the GPU's memory to its computing unit overhead and because we can't do everything in bf16 and at times we have to do math in fp32 (or tf32) we can really expect much less than that. The realistic value will vary from accelerator to accelerator, but for A100 in 2022 getting above 50% (155TFLOPS) was an amazing sustainable throughput for a complex 384 GPUs training setup.

footnote: in 2023 the invention of flash attention and other techniques have pushed the bar to more than 50%.

When we first started tuning things up we were at <100TFLOPS and a few weeks later when we launched the training we managed to get 150TFLOPS.

The important thing to notice here is that we knew that we can't push it further by much and we knew that there was no more point to try and optimize it even more.

So a general rule of thumb for when you prepare for a massive model training - ask around what's the top TFLOPS one can expect to get with a given accelerator on a multi-node setup with the specified precision - and optimize until you get close to that. Once you did stop optimizing and start training.

footnote: For 80GB A100s in 2022 that was 155, in 2023 it has been pushed to about 180TFLOPS.

footnote: When calculating TFLOPS it's important to remember that the math is different if Gradient checkpointing is enabled, since when it's activated more compute is used and it needs to be taken into an account. Usually the cost is of an additional forward path, but recently better methods have been found that saves some of that recomputation.

For decoder transformer models the following is an estimation formula which slightly under-reports the real TFLOPS:

TFLOPS: `model_size_in_B * 4 * 2 * seqlen * global_batch_size / (time_in_sec_per_iteration * total_gpus * 1e3)`

The factor of `4` is used with activation/gradient checkpointing, otherwise it will be `3`. For 100B+ models, activation checkpointing will almost always be on.

So the `3*2` is often called "model FLOPs" and `4*2` - "hardware FLOPs", correlating to MFU and HFU (model and hardware FLOPS per second divided by the accelerator's theoretical peak FLOPS)

```
perl -le '$ng=64; $ms=52; $gbs=1024; $sp=127; $seqlen=2048; print $ms*4*2*$seqlen*$gbs / ( $sp *
$ng * 1e3)'
```

(ng = total gpus, ms = model size in B, gbs = global batch size, sp = seconds per iteration)

Here is the same formula using `bash` env vars and which breaks down GBS into `MBS*DP*GAS` (GAS in this case corresponded to `pp_chunks` which was the number of chunks in the pipeline, but normally GAS just stands for Gradient Accumulation Steps):

```
echo
"($MSIZE*4*2*SEQLEN*$MICRO_BATCH_SIZE*$DP_SIZE*$GAS)/($THROUGHPUT*$NNODES*$GPUS_PER_NODE*1000)" |
bc -l
```

(`$GPUS_PER_NODE` = accelerators per node, so `$NNODES*$GPUS_PER_NODE` is total GPUs - the same role as `$ng` in the Perl one-liner above)

The exact formula is in Equation 3 of Section 5.1 of the Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM paper. You can see the code here.

footnote: For Inference only it'd be: `24Bsh^2 + 4Bs^2h` floating point operations per layer.

Automating FLOP calculation

Manual FLOP calculation, as explained in the previous section, is difficult to get right - many of those formulas have mistakes in them, and many models behave differently depending on various configuration settings. So it can be tricky to get the FLOP count correctly (and across many different model architectures). The PyTorch team developed an automatic way of measuring FLOPs.

```
from torch.utils.flop_counter import FlopCounterMode

flop_counter = FlopCounterMode(mods=model, display=False, depth=None)
with flop_counter:
    model(**input).sum().backward()
total_flops =  flop_counter.get_total_flops()
```

Voila, you have the FLOPs counted for you! In theory, at least. In practice `FlopCounterMode` has proved to have multiple problems impacting performance and memory usage that as of 2026-08 it can't be relied on to replace the manual calculation - so expect to still need the formulas from the previous section, and to treat the counter as a cross-check rather than as the answer.

In my code I run it only on a 2nd iteration (as the first iteration is likely to have some additional compute that is run once). You don't need to repeat it again, you can just cache its value (well, unless you have a situation where iterations aren't the same for some reason).

So all that remains is measuring the time it took each specific iteration to run and dividing FLOPs by time in seconds and `1e12` to get the performance TFLOPS.

```
tflops = total_flops / time / 1e12
```

This will give you a slightly different value on each iteration.

MFU vs HFU

Model FLOPS Utilization (MFU) and Hardware FLOPS Utilization (HFU) estimate how well the hardware is being utilized during `forward` and `backward` passes of the model (including any syncing networking overhead and possibly DataLoader IO).

```
MFU = Estimated_Achieved_FLOPS / Theoretical_FLOPS
HFU = Actual_Achieved_FLOPS / Theoretical_FLOPS
```

HFU measures the actual FLOPS. For example, the technique of Gradient checkpointing/Activation Recomputation repeats all or some parts of the `forward` pass a second time, so factually more FLOS (FLoating point OperationS) are used. Whereas MFU ignores implementation details and accounts only for the theoretical needs of the computation and is thus less accurate.

Reducing Activation Recomputation in Large Transformer Models is a good paper to read about these concepts.

`Theoretical_FLOPS` is what you see on the official accelerator specs. You can find the table of these values for high end accelerators here. So let's use H100 as an example. Its BF16 theoretical TFLOPS is 989TFLOPS.

Now, say, you measured your actual training loop's performance and it was 400TFLOPS as actual achieved FLOPS. Then your HFU is:

```
HFU = 400/989 = 40%
```

If you didn't use activation recomputation feature (not repeating `forward`) your HFU and MFU would be the same. If you did use it, MFU will be lower than HFU, because MFU counts only the theoretical `forward/backward` FLOPs while HFU also counts the extra recomputation FLOPs.

For example Megatron-LM published the following stats for A100-80GB:

Model Size	Model FLOPs Utilization	Hardware FLOPs Utilization
22B	41.5%	43.7%
175B	51.4%	52.8%
530B	56.0%	57.0%
1T	56.3%	57.0%

As you can see, since Megatron-LM was using activation recomputation for these trainings, MFU < HFU.

More recent H100+A100 MFU/HFU numbers have been published here.

Now, whenever you see MFU or HFU numbers published you have to be careful comparing those numbers to any other similar numbers, until you know that the same way was used to calculate FLOPS. Since `HFU=Actual_Achieved_FLOPS/Theoretical_FLOPS` and `Theoretical_FLOPS` are fixed, the only variable here is the `Actual_Achieved_FLOPS` and since

most of the time an estimated value is calculated based on parameter shapes, there are many versions of calculations out there, some of which are slightly imprecise whereas others are very imprecise. Compilers may also impact the effective FLOPs, by optimizing some operations away. Moreover you don't know how iteration time was measured.

To recall `TFLOPS = FLOS / iteration_duration`. So, in order to do a fair comparison the 2 main questions to ask are:

1. Was the total used floating point operations calculated in the same way?
2. Was the time component calculated back-to-back of each iteration, including the `DataLoader` and logging, vs only `fwd+bwd` parts.

If either one or both of these mismatch then you can't make a fair comparison.

Unfortunately, most of the time papers and blog posts just report the MFU number w/o a link to how it was calculated.

But, do not fear, if you have trouble comparing your results with competing results, remember the measurement artifacts described above. These artifacts do not improve the bottom-line throughput, thus, as long as you consistently use whatever way you choose to calculate TFLOPS, you will immediately see when your application's performance has improved or degraded, as relative numbers are most important for you.

MFU is a very rough approximation

Most of the training/finetuning regimes use a mixed precision, yet when MFU/HFU are calculated the fastest format's compute is used. For example, for a BF16-mixed precision training some parts of the compute are done in BF16, yet others in FP32! but we measure the FLOPS as if everything was done in BF16, which, of course, leads to a very imprecise measurement. Ideally, each segment of the compute will be measured separately to account to when which format was used. The reason it sort of works is because the smaller format compute usually dominates those mixed precision training regimes.

Moreover, depending on the cluster setup - in particular storage IO and network IO are heavily involved, the same software may deliver different MFUs, because not all clusters are created equal. Therefore it's OK to compare your particular setup's MFU before and after the optimization, but it's very difficult to compare your setup's MFU to another team's cluster's MFU.

How To Improve Speed and Save Memory

The more GPU memory you have for your batch size (BS) the more efficient the GPUs will be at performing compute, and the faster you will complete your task since you will be able to go through data faster.

Of course, this section is crucial for when you get GPU OOM with even BS=1 and you don't want to rent/buy more hardware.

Here is an overview of what features can help to either improve speed or save memory

Method	Speed	Memory
Gradient accumulation	Yes	Yes
Gradient checkpointing	No*	Yes
Mixed precision training	Yes	No
Batch size	Yes	Yes
Optimizer choice	Yes	Yes
DataLoader	Yes	No
DeepSpeed ZeRO	No	Yes
Flash Attention	Yes	Yes

* Gradient checkpointing slows things down for the given batch size, but since it frees up a lot of memory, enabling a much larger BS, it actually improves the overall speed.

Anatomy of Model's Operations

Transformers architecture includes 3 main groups of operations grouped below by compute-intensity.

1. **Tensor Contractions**

   Linear layers and components of Multi-Head Attention all do batched **matrix-matrix multiplications**. These operations are the most compute-intensive part of training a transformer.

2. **Statistical Normalizations**

   Softmax and layer normalization are less compute-intensive than tensor contractions, and involve one or more **reduction operations**, the result of which is then applied via a map.

3. **Element-wise Operators**

   These are the remaining operators: **biases, dropout, activations, and residual connections**. These are the least compute-intensive operations.

This knowledge can be helpful to know when analyzing performance bottlenecks.

This summary is derived from [Data Movement Is All You Need: A Case Study on Optimizing Transformers 2020]

Anatomy of Model's Memory Usage

We've seen that training the model uses much more memory than just putting the model on the GPU. This is because there are many components during training that use GPU memory. The components on GPU memory are the following:

1. model weights
2. optimizer states
3. gradients
4. forward activations saved for gradient computation
5. temporary buffers
6. functionality-specific memory

A typical model trained in mixed precision with AdamW requires 18 bytes per model parameter plus activation memory and temp memory.

Let's look at the details.

Model Weights:

- 4 bytes * number of parameters for fp32 training
- 6 bytes * number of parameters for mixed precision training (maintains a model in fp32 and one in fp16/bf16 in memory)

Optimizer States:

- 8 bytes * number of parameters for normal AdamW (maintains 2 states)
- 4 bytes * number of parameters for AdamW running at bf16. See this work that uses `AnyPrecisionAdamW`.
- 4 bytes * number of parameters for optimizers like SGD with momentum (maintains only 1 state) or LION, or Adafactor (and others) (Adafactor uses some additional memory beside 4 bytes)
- 2 bytes * number of parameters for 8-bit AdamW optimizers like bitsandbytes

Gradients

- 4 bytes * number of parameters for either fp32 precision and in some frameworks with mixed half-precision training.

- 2 bytes * number of parameters for non-mixed half-precision and in some frameworks with mixed half-precision training.

Forward Activations

- size depends on many factors, the key ones being sequence length, hidden size and batch size.

There are the input and output that are being passed and returned by the forward and the backward functions and the forward activations saved for gradient computation.

Here is a rough breakdown of activation memory per GPU:

1. Activation working memory: `dtype_size * num_of_hidden_states_copies * bs * seqlen * hidden_size`
2. Activation checkpoints memory: `dtype_size * num_layers * bs * seqlen * hidden_size`
3. Logits working memory: `fp32_size * bs * seqlen * vocab_size`

In all the following calculations:

- `dtype_size`: size of the one element of `dtype` being used, e.g. `1` byte for fp8, `2` for bf16, `4` for fp32, etc. - see Tensor precision / Data types for the full set
- `fp32_size`: `4` bytes
- `bs`: batch size
- `hidden_size`: hidden size of the model
- `seqlen`: the sequence length
- `num_layers`: the number of layers the model has
- `vocab_size`: vocabulary size

And `dtype_size * bs * seqlen * hidden_size` is the size of the `hidden_states` tensor, because that's its shape `[bs, seqlen, hidden_size]` multiplied by the size in bytes of a single element.

In the following explanation let's use the Llama-3 architecture and its 8B configuration:

```
hidden_size=4096
num_layers=32
vocab_size=128256
```

and let's use

```
dtype=bf16 (2 bytes)
bs=1
seqlen=32768
```

We can immediately calculate the size of `hidden_states` tensor in bf16 to be: `2 * 1 * 32768 * 4096 / 2**30 = 0.25GiB`

Part 1. So first let's calculate how much working memory is needed to process a single layer forward:

`activation_memory_per_layer = num_of_hidden_states_copies * dtype_size * bs * seqlen * hidden_size`

The `num_of_hidden_states_copies` is typically between 20 and 50 and will vary between different model architectures. To get a good feeling for the variations - let's measure a handful of models using activation-memory-per-layer.py. What we get is:

```
24.0 "HuggingFaceTB/SmolLM2-360M"
```

```
28.0 "meta-llama/Llama-3.1-8B-Instruct"
28.0 "mistralai/Mistral-7B-Instruct-v0.3"
29.8 "deepseek-ai/DeepSeek-R1-0528-Qwen3-8B"
38.0 "Qwen/Qwen3-4B"
48.0 "google/gemma-1.1-2b-it"
```

So it's easy to see that Gemma makes two times more copies of `hidden_states` than SmolLM2 to compute a single layer forward.

It'd be difficult to compare the models exactly if their `hidden_size` isn't the same, but comparing the `num_of_hidden_states_copies` co-efficient is easy.

Clearly some architectures allocate additional memory besides copies of `hidden_states` as can be seen with `deepseek-ai/DeepSeek-R1-0528-Qwen3-8B` whose co-efficient isn't an integer.

Now to our example model:

`activation_memory_per_layer = 2 * 28 * 1 * 32768 * 4096 / 2**30 = 7GiB` (28 came from the measurements output a few paragraphs earlier for `meta-llama/Llama-3.1-8B-Instruct`).

Part 2. Now that we know the memory allocated by each layer, 2 possible things will happen next depending on whether gradient checkpointing is enabled or not:

- If it's enabled, then we need to store `num_layers * hidden_states`.

In our example that would be: `32 * 0.25 = 8GiB` (we calculated `hidden_states = 0.25GiB` earlier).

- If it's not enabled, then activation memory will grow to `num_layers * activation_memory_per_layer`.

In our example that would be: `32 * 7 = 224GiB`

Notice 8GiB vs 224GiB - that's a huge difference - therefore, clearly everybody activates the gradient checkpointing feature despite the cost of recalculating the forward pass a second time.

Part 3. Finally we have the loss calculation which typically requires manifesting the logits in fp32 and here the `vocab_size` dimension is typically the biggest dimension, hence: `fp32_size * bs * seqlen * vocab_size`

In our example that would be: `4 * 1 * 32768 * 128256 / 2**30 = 15.7GiB` (~16GiB)

As you can see logits will use more memory than all the checkpointing activations for this model.

Moreover, practically you typically end up with a factor of 6 instead of 4 for `dtype_size`, since usually `logits` is first created in the lower precision and then upcast to fp32 so you have 2 copies of it, thus `2+4=6`.

Part 4. Total required activation memory is the sum of memory needed to compute one layer forward + activation checkpoints + logits activation memory when using gradient checkpointing. When not using gradient checkpointing it's the sum of 2 and 3, i.e. all the forwards activations and the logits memory.

In our example:

- w/ gradient checkpointing: `7 + 8 + 16 = 31GiB`
- w/o gradient checkpointing: `224 + 16 = 240GiB`

Temporary Memory

Additionally there are all kinds of temporary variables which get released once the calculation is done, but in the moment these could require additional memory and could push to OOM. Therefore when coding it's crucial to think strategically about such temporary variables and sometimes to explicitly free those as soon as they are no longer needed.

Functionality-specific memory

Then your software could have special memory needs. For example, when generating text using beam search, the software

needs to maintain multiple copies of inputs and outputs.

For **inference**, the math is very similar to training, minus optimizer states and gradients. And for model weights there is just a single multiplier of the number of parameters:

- 4 bytes in fp32
- 2 bytes in fp16/bf16
- 1 byte in fp8/int8
- 0.5 bytes in fp4/int4

See Model Weights in the inference chapter for lower-precision and quantized formats, including fp8, fp6, fp4, integer quantization, and MX formats.

Another excellent resource that takes you through the memory needs and other requirements is Transformer Math 101.

The EAI cookbook contains a set of calculation scripts that can output the theoretical memory overhead for a given training or inference calculation run based on your configuration and setup.

There is a very handy GPU VRAM Estimator from Alexander Smirnov, and the notes to how it works.

Additional GPU memory usage

In addition to the memory usage described in the previous section, there are other consumers of the GPU memory - so you never get the full memory for your model's use.

Preloaded CUDA kernels memory usage

When PyTorch uses CUDA for the first time, it may use up 0.5-2GiB of GPU memory, reducing the GPU's total available memory. This memory won't be accounted for by torch memory profiler.

The size of allocated memory for cuda kernels varies between different GPUs, and also it can be different between pytorch versions. Let's allocate a 4-byte tensor on cuda and check how much GPU memory is used up upfront.

With `pytorch==1.10.2`:

```
$ CUDA_MODULE_LOADING=EAGER python -c "import torch; x=torch.ones(1).cuda(); free, total =
map(lambda x: x/2**30, torch.cuda.mem_get_info()); \
used=total-free; print(f'pt={torch.__version__}: {used=:0.2f}GiB, {free=:0.2f}GiB,
{total=:0.2f}GiB')"
pt=1.10.2: used=1.78GiB, free=77.43GiB, total=79.21GiB
```

With `pytorch==1.13.1`:

```
$ CUDA_MODULE_LOADING=EAGER python -c "import torch; x=torch.ones(1).cuda(); free, total =
map(lambda x: x/2**30, torch.cuda.mem_get_info()); \
used=total-free; print(f'pt={torch.__version__}: {used=:0.2f}GiB, {free=:0.2f}GiB,
{total=:0.2f}GiB')"
pt=1.13.1: used=0.90GiB, free=78.31GiB, total=79.21GiB
```

The older pytorch "wasted" 1.78GiB of an A100 where the newer used only 0.9GiB - and 0.9GiB recovered can be the saving grace in an OOM situation.

Do not read that as a one-way trend, though. This overhead is a side effect of how a given pytorch and CUDA version happen to package and load their kernels, so it moves in both directions between releases: it nearly halved from `1.10.2` to `1.13.1`,

then edged back up slightly by `2.1.1` below. The figures here are from 2022-2023 and are kept because they show how wide the swing can be - close to a gigabyte on the same GPU purely from a version change. Treat them as an illustration of the effect rather than as numbers to plan against, and run the one-liner above on your own combination of GPU, pytorch and CUDA to get the figure that actually applies to you.

`CUDA_MODULE_LOADING=EAGER` is needed in the recent pytorch version if we want to force cuda kernels pre-loading, which are otherwise lazy-loaded on demand. But do not use this setting in production since it's likely to use more memory than needed. The whole point of lazy-loading is to load only the kernels that are needed.

With `pytorch==2.1.1`:

```
$ CUDA_MODULE_LOADING=EAGER python -c "import torch; x=torch.ones(1).cuda(); free, total =
map(lambda x: x/2**30, torch.cuda.mem_get_info()); \
used=total-free; print(f'pt={torch.__version__}: {used=:0.2f}GiB, {free=:0.2f}GiB,
{total=:0.2f}GiB')"
pt=2.1.1+cu121: used=0.92GiB, free=78.23GiB, total=79.15GiB
```

As compared to the lazy mode:

```
$ python -c "import torch; x=torch.ones(1).cuda(); free, total = map(lambda x: x/2**30,
torch.cuda.mem_get_info()); \
used=total-free; print(f'pt={torch.__version__}: {used=:0.2f}GiB, {free=:0.2f}GiB,
{total=:0.2f}GiB')"
pt=2.1.1+cu121: used=0.47GiB, free=78.68GiB, total=79.15GiB
```

There is a 450MiB difference, but here we only loaded kernels to do `torch.ones` - the actual memory allocated at run time with other code using torch API will be somewhere between 0.47 and 0.92GiB.

`torch.distributed` memory usage

When using `torch.distributed` expect ~1-2GiB of GPU memory taken away just to initialize things - the more GPUs the higher the memory used. Different backends are likely to use a different amount of memory. And this memory won't be accounted for by torch memory profiler.

Here is torch-dist-mem-usage.py that demonstrates the actual memory usage:

```
$ python -u -m torch.distributed.run --nproc_per_node=2 --rdzv_endpoint localhost:6000
--rdzv_backend c10d \
torch-dist-mem-usage.py
[...]
[0] mp: before barrier
[0] mp: MA 0.00 GiB | Max_MA 0.00 GiB | CA 0.00 GiB | Max_CA 0.00 GiB | NV 0.55 GiB | CPU Virtual
Memory:  used = 68.79 GiB, percent = 3.4%
[0] mp: after barrier
[0] mp: MA 0.00 GiB | Max_MA 0.00 GiB | CA 0.00 GiB | Max_CA 0.00 GiB | NV 2.00 GiB | CPU Virtual
Memory:  used = 69.33 GiB, percent = 3.5%
[0] mp: after 2nd barrier
[0] mp: MA 0.00 GiB | Max_MA 0.00 GiB | CA 0.00 GiB | Max_CA 0.00 GiB | NV 2.00 GiB | CPU Virtual
Memory:  used = 69.33 GiB, percent = 3.5%
[0] mp: after dist destroy
```

```
[0] mp: MA 0.00 GiB | Max_MA 0.00 GiB | CA 0.00 GiB | Max_CA 0.00 GiB | NV 1.28 GiB | CPU Virtual
Memory:  used = 69.25 GiB, percent = 3.5%
[0] mp: after dist destroy
[0] mp: MA 0.00 GiB | Max_MA 0.00 GiB | CA 0.00 GiB | Max_CA 0.00 GiB | NV 1.28 GiB | CPU Virtual
Memory:  used = 69.25 GiB, percent = 3.5%
```

So you can see that the CUDA kernels took 0.55GiB (first line `NV 0.55 GiB`), but when `dist.barrier` was run an additional 1.5GiB were consumed. If you run `init_process_group` with `device_id` arg then the additional memory allocation will move to that call instead and less will be used by `dist.barrier`.

Choosing between `nccl`, `nccl2` and `nccl-lazy`

Starting from PyTorch 2.14, PyTorch's distributed package includes a second implementation over the same NCCL library. Despite its name, `nccl2` is not NCCL version 2 and does not replace NCCL's collective algorithms. What it changes is how PyTorch creates, splits, waits for and aborts the communicators those collectives run on. In PyTorch 2.14, `init_process_group("nccl")` still selects the established `ProcessGroupNCCL`; select the new implementation explicitly with `backend="nccl2"`, or set `TORCH_DIST_USE_NCCL2=1` to make `"nccl"` resolve to it. The explicit `nccl-legacy` name always selects the established implementation, and `nccl-lazy` is a variant of `nccl2` that creates a communicator per peer on first use rather than up front.

Measured on an idle 8x H200 node with PyTorch 2.14.0, CUDA 13.0 and NCCL 2.30.7, a 1GiB `all_reduce` landed between 458 and 466GBps `busbw` under both, and repeats of a single implementation varied by as much as the gap between the two - so these runs show no throughput difference. Both issue the same NCCL collectives.

The measurable difference was memory. After `init_process_group` and the first `dist.barrier`, rank 0's accelerator held 1.56GiB under `nccl` and 5.16GiB under `nccl2` - reproduced across runs, and unchanged by whether `device_id` was passed. So don't switch expecting faster collectives, and measure the memory cost on your own topology before adopting it. What `nccl2` offers instead is control over the communicator's lifetime: subgroups built at `new_group` time rather than at their first collective, and - through APIs the release marks as unstable as of 2.14 - a group that can be rebuilt in place after a rank dies instead of requiring a restart.

`nccl-lazy` is for when each rank talks to only a few peers, as in Pipeline Parallelism, since a communicator that is never used is never built. It does not lower peak memory for a group that runs collectives from the first step, because that communicator gets built either way.

Reference: PyTorch 2.14 release blog.

Memory fragmentation

As the model allocates and frees tensors, the GPU memory could fragment. That is there could be enough free memory to allocate, say, 1GiB of contiguous memory, but it could be available in 100s of small segments spread out through the memory and thus even though the memory is available it can't be used unless very small allocations are made.

Environment variable `PYTORCH_ALLOC_CONF` comes to help and allows you to replace the default memory allocation mechanisms with more efficient ones. For more information see Memory management. I found `PYTORCH_ALLOC_CONF=expandable_segments:True` to be extremely helpful when the code performs a lot of tensor reshaping, which would normally massively fragment the GPU memory.

Batch sizes

First, there are usually two batch sizes:

1. micro batch size (MBS), also known as batch size per gpu - this is how many samples a single gpu consumes during a model's single `forward` call.
2. global batch size (GBS) - this is the total amount of samples consumed between two optimizer steps across

all participating GPUs.

Model replica is how many gpus are needed to fit the full model.

- If the model fits into a single GPU a model replica takes 1 GPU. Usually then one can use multiple GPUs to perform Data Parallelism
- If the model doesn't fit into a single GPU, it'd usually require some sort of sharding technique - it can be Tensor Parallelism (TP), Pipeline Parallelism (PP), or ZeRO Data Parallelism (ZeRO-DP).

You can have as many data streams as there are replicas. Which is the same as the value of DP.

- So in the simple case of a model fitting into a single GPU. There are as many data streams as there are GPUs. DP=N_GPUS
- when the model doesn't fit onto a single GPU, then `DP=N_GPUs/(TP*PP)` in the case of 3D parallelism and DP=ZeRO-DP in the case of ZeRO parallelism.

Going back to our global batch size (GBS) it's calculated as:

```
GBS = MBS*DP
```

So if you have 8 gpus (N_GPUS=8) and your MBS=4 and you do DP you end up with having GBS=32 because:

```
GBS = MBS*DP = 4*8 = 32
```

If you use TP with a degree of 2 (TP=2) and PP with a degree of 2 (PP=2) this means each model replica takes 4 gpus (`TP*PP`), and thus with N_GPUS=8

```
DP = N_GPUS/(TP*PP) = 8 / (2*2) = 2
```

and now GBS becomes:

```
GBS = MBS*DP = 4*2 = 8
```

If your training setup requires Gradient Accumulation, one usually defines the interval of how many steps to wait before performing a gradient accumulation. The term is usually Gradient Accumulation Steps (GAS). If GAS=4 (i.e. sync grads every 4 steps) and TP=1, PP=1 and DP=8:

```
DP = N_GPUS/(TP*PP) = 8 / (1*1) = 8
GBS = MBS*DP*GAS = 4*8*4 = 128
```

Typically you want to make the micro batch size as large as possible so that the GPU memory is close to being full, but not too full.

With large models usually there is not much free GPU memory left to have a large micro batch size, therefore every additional sample you can fit is important.

While it's super important that sequence length and hidden size and various other hyper parameters are high multiples

of 2 (64, 128 and higher) to achieve the highest performance, because in most models the batch dimension is flattened with the sequence length dimension during the compute the micro batch size alignment usually has little to no impact on performance.

Therefore if you tried to fit a micro batch size of 8 and it OOM'ed, but 7 fits - use the latter rather than 4. The higher the batch size the more samples you will be able to fit into a single step.

Of course, when using hundreds of GPUs your global batch size may become very large. In that case you might use a smaller micro batch size or use less GPUs or switch to a different form of data parallelism so that the GPUs work more efficiently.

Gradient Accumulation

The idea behind gradient accumulation is to instead of calculating the gradients for the whole batch at once to do it in smaller steps. The way we do that is to calculate the gradients iteratively in smaller batches by doing a forward and backward pass through the model and accumulating the gradients in the process. When enough gradients are accumulated we run the model's optimization step. This way we can easily increase the overall batch size to numbers that would never fit into the GPU's memory. In turn, however, the added forward and backward passes can slow down the training a bit.

Gradient Accumulation Steps (GAS) is the definition of how many steps are done w/o updating the model weights.

When using Pipeline parallelism a very large Gradient Accumulation is a must to keep the pipeline's bubble to the minimum.

Since the optimizer step isn't performed as often with gradient accumulation there is an additional speed up here as well.

The following benchmarks demonstrate how increasing the gradient accumulation steps improves the overall throughput (20-30% speedup):

- RTX-3090
- A100

When data parallelism is used gradient accumulation further improves the training throughput because it reduces the number of gradient reduction calls, which is typically done via the `all_reduce` collective which costs a 2x size of gradients to be reduced. So for example, if one goes from GAS=1 to GAS=8 in `DistributedDataParallelism` (DDP) the network overhead is reduced by 8x times, which on a slow inter-node network can lead to a noticeable improvement in the training throughput.

Gradient Checkpointing

Gradient Checkpointing is also known as **Activation Recomputation** and **Activation Checkpointing**.

This methodology is only relevant for training, and not during inference.

Enabling gradient checkpointing allows one to trade training throughput for accelerator memory. When this feature is activated instead of remembering the outputs of, say, transformer blocks until the `backward` pass is done, these outputs are dropped. This frees up huge amounts of accelerator memory. But, of course, a `backward` pass is not possible without having the outputs of `forward` pass, and thus they have to be recalculated.

This, of course, can vary from model to model, but typically one pays with about 20-25% decrease in throughput, but since a huge amount of gpu memory is liberated, one can now increase the batch size per gpu and thus overall improve the effective throughput of the system. In some cases this allows you to double or quadruple the batch size if you were already able to do a small batch size w/o OOM. (Recent papers report as high as 30-40% additional overhead.)

For example, in HF Transformers models you do `model.gradient_checkpointing_enable()` to activate it in your custom Trainer or if you use the HF Trainer then you'd activate it with `--gradient_checkpointing 1`.

Also check out this the paper: Reducing Activation Recomputation in Large Transformer Models which describes a way to avoid most activation recomputations and thus save both memory and compute.

Memory-efficient optimizers

The most common optimizer is Adam. It and its derivatives all use 8 bytes per param (2x fp32 tensors - one for each momentum), which account for almost half the memory allocation for the model, optimizer and gradients. So at times using other optimizers may save the day, if they successfully train that is. Not all optimizers are suitable for all training tasks.

4-byte optimizers:

- There are optimizers like Adafactor that need only 4 bytes. Same goes for the LION optimizer.
- `AnyPrecisionAdamW`. Some courageous souls try to do the whole training in BF16 (not mixed precision!), including the optimizer and thus need only 4 bytes per parameter for optim states. See this work. Hint: this optimizer requires Kahan summation and/or stochastic rounding, see Revisiting BFloat16 Training (2020). You need only 8 bytes per parameter for weights, optim states and gradients here! Instead of 18!

2-byte optimizers:

- There are quantized solutions like `bnb.optim.Adam8bit` which uses only 2 bytes instead of 8 (1 byte per momentum). You can get it from here. Once installed, if you're using HF Trainer, you can enable it on with just passing `--optim adamw_bnb_8bit`!

For speed comparisons see this benchmark Speed-wise:`apex`'s `apex.optimizers.FusedAdam` optimizer is so far the fastest implementation of Adam. Since pytorch-2.0 torch.optim.AdamW added support for `fused=True` option, which brings it almost on par with `apex.optimizers.FusedAdam`.

Model execution speed

`forward` vs `backward` Execution Speed

For convolutions and linear layers there are 2x flops in the backward compared to the forward, which generally translates into ~2x slower (sometimes more, because sizes in the backward tend to be more awkward). Activations are usually bandwidth-limited, and it's typical for an activation to have to read more data in the backward than in the forward (e.g. activation forward reads once, writes once, activation backward reads twice, `gradOutput` and output of the forward, and writes once, `gradInput`).

Input data affects measured speed

Measured step time depends on the *values* in the tensors, not only on their shapes. High-entropy inputs flip more transistors per cycle, draw more power, and can pull the boost clock down under the same power / thermal envelope that low-entropy or all-zero inputs leave alone - so an identical kernel may complete its compute at different times than on low-entropy or all-zero inputs of the same shape. For a training or inference bench this means: seed the RNG that builds the inputs (`torch.manual_seed(...)`), and feed a realistic distribution rather than zeros. The effect is small when the distribution is held fixed, but it is not zero. Full write-up, including the power and denormal mechanisms and the citations, under Input data affects measured performance. This is about *timing* conditions; for *numerical* determinism see Reproducibility.

Vector and matrix size divisibility

The paper, The Case for Co-Designing Model Architectures with Hardware investigates the effects of transformer sizing on the underlying hardware. The associated scripts allow you to run the benchmarks yourself if you're running on hardware besides NVIDIA V100/A100.

One gets the most efficient performance when batch sizes and input/output neuron counts are divisible by a certain number, which typically starts at 8, but can be much higher as well. That number varies a lot depending on the specific

hardware being used and the dtype of the model.

For fully connected layers (which correspond to GEMMs), NVIDIA provides recommendations for input/output neuron counts and batch size.

Tensor Core Requirements define the multiplier based on the dtype and the hardware. For example, for fp16 a multiple of 8 is recommended, but on A100 it's 64!

For parameters that are small, there is also Dimension Quantization Effects to consider, this is where tiling happens and the right multiplier can have a significant speedup.

The Case for Co-Designing Model Architectures with Hardware provides much greater detail on tile/wave quantization and the number of attention heads, but the highlights are:

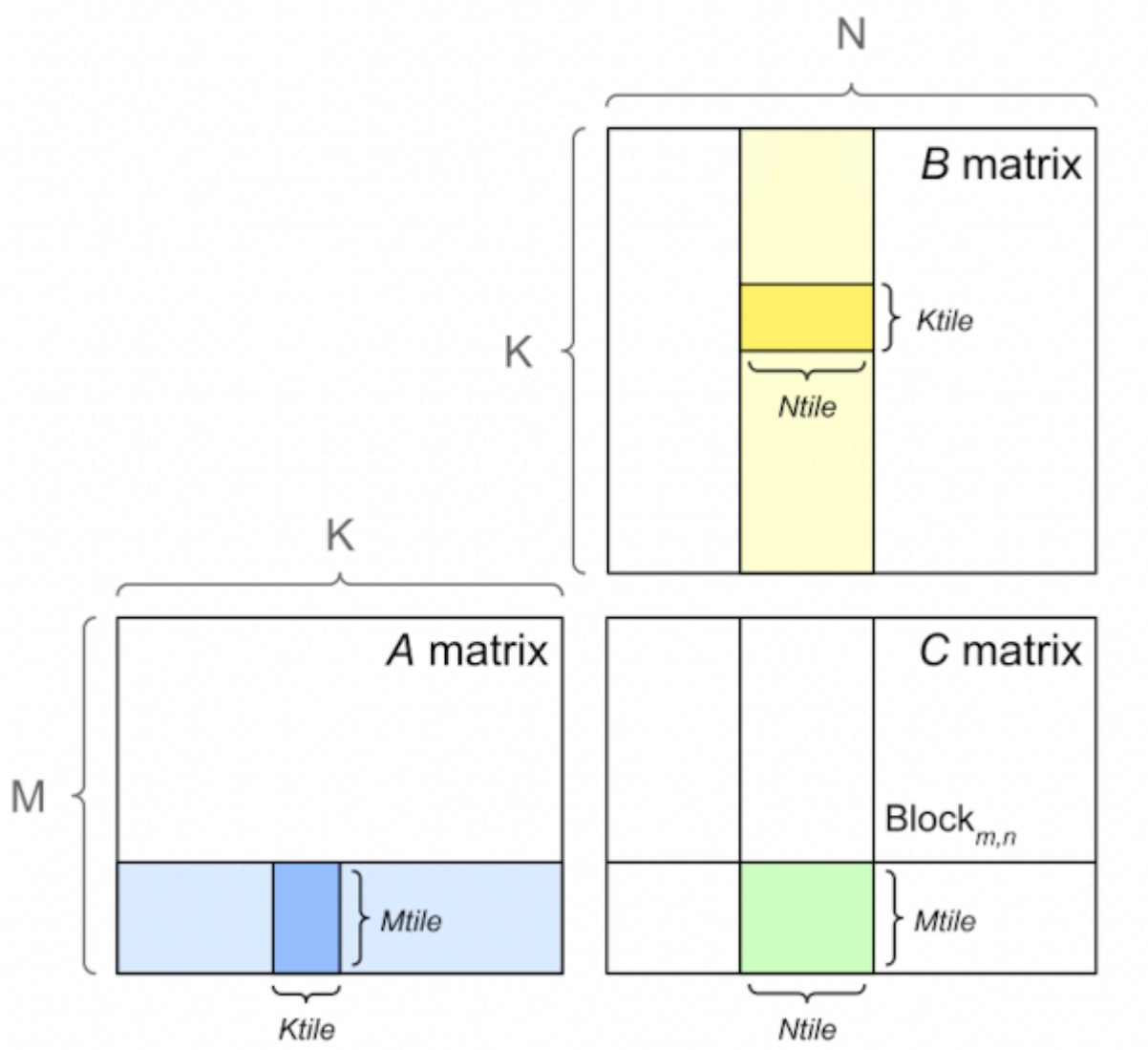
Tile and wave quantization

Notation:

- `a`: Number of attention heads
- `h`: Hidden dimension size
- `s`: Sequence length
- `b`: Microbatch size
- `t`: Tensor-parallel size

First, some background.

NVIDIA GPUs divide the output matrix into regions or tiles as shown in the below figure and schedule them to one of the available streaming multiprocessors (SM) on the GPU (e.g., A100 GPUs have 108 SMs). Each tile or thread block is processed in a Tensor Core, which NVIDIA introduced for fast tensor operations. NVIDIA Tensor Cores are only available for GEMMs with appropriate dimensions. Tensor Cores can be fully utilized when GEMM dimensions `m`, `k`, and `n` are multiples of 16 bytes and 128 bytes for V100 and A100 GPUs, respectively. Since a FP16 element is 2 bytes, this corresponds to dimension sizes that are multiples of 8 and 64 elements, respectively. If these dimension sizes are not possible, Tensor Cores perform better with larger multiples of 2 bytes.

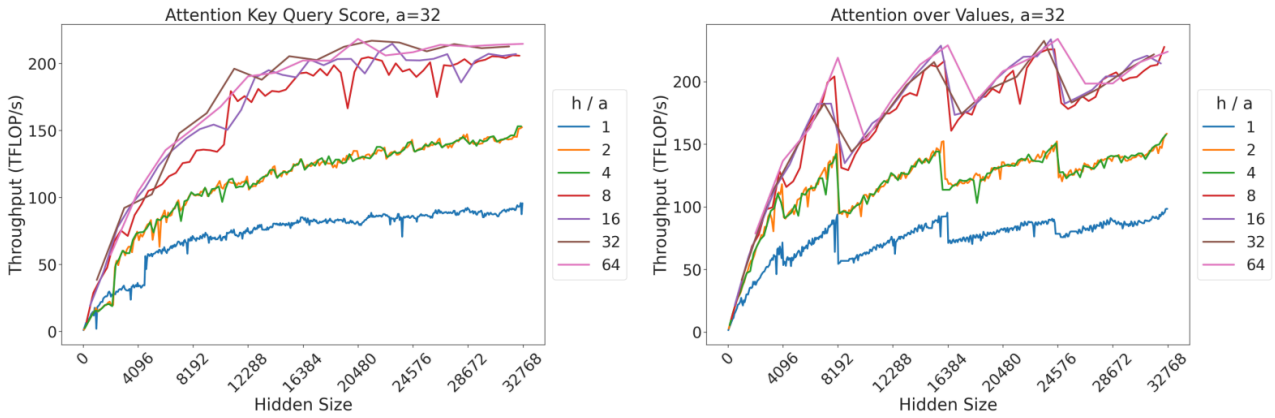


There are multiple tile sizes that the kernel can choose from. If the GEMM size does not divide evenly into the tile size, there will be wasted compute, where the thread block must execute fully on the SM, but only part of the output is necessary. This is called the **tile quantization** effect, as the output is quantized into discrete tiles.

Another quantization effect is called **wave quantization**. As the thread blocks are scheduled to SMs, only as many thread blocks as there are SMs may be scheduled at a time - 108 on A100, and 132 on H100/H200 (measured with `torch.cuda.get_device_properties(0).multi_processor_count`, so substitute your own accelerator's count before using any of the numbers here). Taking A100's 108: if 109 thread blocks must be scheduled, two rounds, or waves, of thread blocks must be scheduled to GPU. The first wave will have 108 thread blocks, and the second wave will have 1. The second wave will have almost the same latency as the first, but with a small fraction of the useful compute. As the matrix size increases, the last or tail wave grows. The throughput will increase, until a new wave is required. Then, the throughput will drop.

What this means for transformers, is that for a given ratio of `h/a`, one needs to ensure they're on the crest of a wave. If you're using NVIDIA V100/A100 GPUs, we've already done this work for you in https://arxiv.org/pdf/2401.14489

An example of this for 32 attention heads:



(a) Attention key-query score GEMM throughput for 32 attention heads. (b) Attention over value GEMM throughput for 32 attention heads.

Fig. 7: Attention GEMM performance on A100 GPUs. Each plot is a single series (i.e. if we didn't split, there would be three regions with spikes), but split by the largest power of two that divides h/a to demonstrate that more powers of two leads to better performance up to $h/a = 64$.

More powers of 2 in `h/a` helps!

Do more SMs give more TFLOPS?

Take two accelerators identical for a given dtype - same matmul units per SM, same FMAs per clock, same clock - where one has more SMs. Theoretical peak scales linearly with the count, because that is how it is derived - `clock * FMAs_per_clock_cycle_per_compute_unit * 2 * num_compute_units` - and that last term is a fixed multiple of the SM count: 4 Tensor Cores per SM on NVIDIA's Ampere, Hopper and Blackwell, with AMD and Intel counting their own units. Only the fixedness matters here. Achievable throughput does *not* scale linearly, and the gap is wave quantization.

A GEMM decomposes into `C` thread blocks, `C = ceil(m/Tm) * ceil(n/Tn)` for a `Tm x Tn` tile. On `S` SMs that takes `W = ceil(C/S)` waves, and since a wave costs the same whether it is full or not, time is proportional to `W`. So for `S1 < S2` the speedup is not `S2/S1` but:

```
speedup = ceil(C/S1) / ceil(C/S2)
```

gives us a plot, which is a staircase, not a line. For example, B300 ships in at least two SKUs, with **148** and **152** SMs - same silicon, same clock, an SM ratio of `152/148` = **1.027**:

thread blocks	waves @148	waves @152	speedup	what happens
148	1	1	1.000x	one wave on both - 4 SMs idle
152	2	1	2.000x	crossover - 2x from 2.7% more SMs
296	2	2	1.000x	no gain - 296 = 2 x 148 fills 148 exactly
304	3	2	1.500x	crossover again
3584	25	24	1.042x	a real transformer MLP shape, see below
50000	338	329	1.027x	asymptote at the 1.027x SM ratio

footnote: thread blocks are also known as Cooperative Thread Array (CTA)

2.7% more SMs buys anywhere from **nothing** to **2x**, on shape alone. Below one wave the extra SMs have no work. At a crossover the narrower part pays for a nearly-empty second wave and loses by up to 2x. Once both need the same wave count they tie again, and now the *wider* part wastes more: at `C = 296` the 148-SM part packs two waves exactly while the 152-SM part idles 8 slots in its second. Only as `C` grows does the advantage settle on the SM ratio - still 1.4% off at 3584, within 0.03% by 50000 - which is why vendors quote peak on enormous matmuls, the one place the staircase is flat.

The `3584` row is a shape you actually run: the MLP up-projection of a Llama-3.1-8B forward pass at 8K tokens. `hidden_size=4096` and `intermediate_size=14336` give an `8192 x 14336` output, which over a `128 x 256` tile is `64 * 56` = 3584 blocks, once per layer for 32 layers. It lands *above* the 1.027x ratio - the staircase straddles the ratio rather than approaching it from below.

So **an SM count does not tell you a throughput advantage** without your shapes, and a shape tuned on one part can land on the wrong side of a crossover on another. To place yourself, count your blocks and divide by your own SM count.

A published peak is an **aggregate over the whole part**, never a per-SM rate: B300's 2250TFLOPS bf16 is what all 148 SMs deliver together, about 15.2 each. A single-wave matmul is bounded by that per-SM rate alone.

And the count is SKU-dependent, so the published peak moves with it. NVIDIA gives 160 SMs for "the full GPU implementation", with the caveat that "available SM count and HBM capacity varies by SKU" - the same gap the [cache table] records a generation back, `GH100 full implementation` at 144 against `H100 SXM`'s 132. Both measured B300 SKUs ship below the full die, and a B200 SKU also reads 148. Dividing each published figure by 148 and scaling to 152 gives what the other SKU is entitled to publish:

dtype	published	per SM (/148)	at 152 SMs
bf16	2250	15.20	2311
fp8	4500	30.41	4622
fp4	12600	85.14	12941
nvfp4	15000	101.35	15405

Every dtype moves by that same 1.027x, which leaves a corollary: **NVIDIA publishes one set of numbers for `B300 SXM`, and two real SKUs cannot both match them.** If 2250 belongs to the 148-SM part then the 152-SM part is entitled to 2311; if it belongs to the 152-SM part then the 148-SM one reaches only 2191. So a peak quoted without its SM count is approximate for at least one of the parts sold under that name. Note the asymmetry: that 2.7% gap in *published* peak is exact, while the gap in *achievable* throughput is the staircase above - nothing, 2x, or 2.7% depending entirely on shape.

To get the SM count of your accelerator:

```
python -c "import torch; p=torch.cuda.get_device_properties(0); print(p.name,
p.multi_processor_count)"
```

Number and size of attention heads

Generally, it's most computationally efficient to keep the ratio of `h/a` as large as possible without accuracy degradation. A good figure from [The Case for Co-Designing Model Architectures with Hardware] showing this effect is:

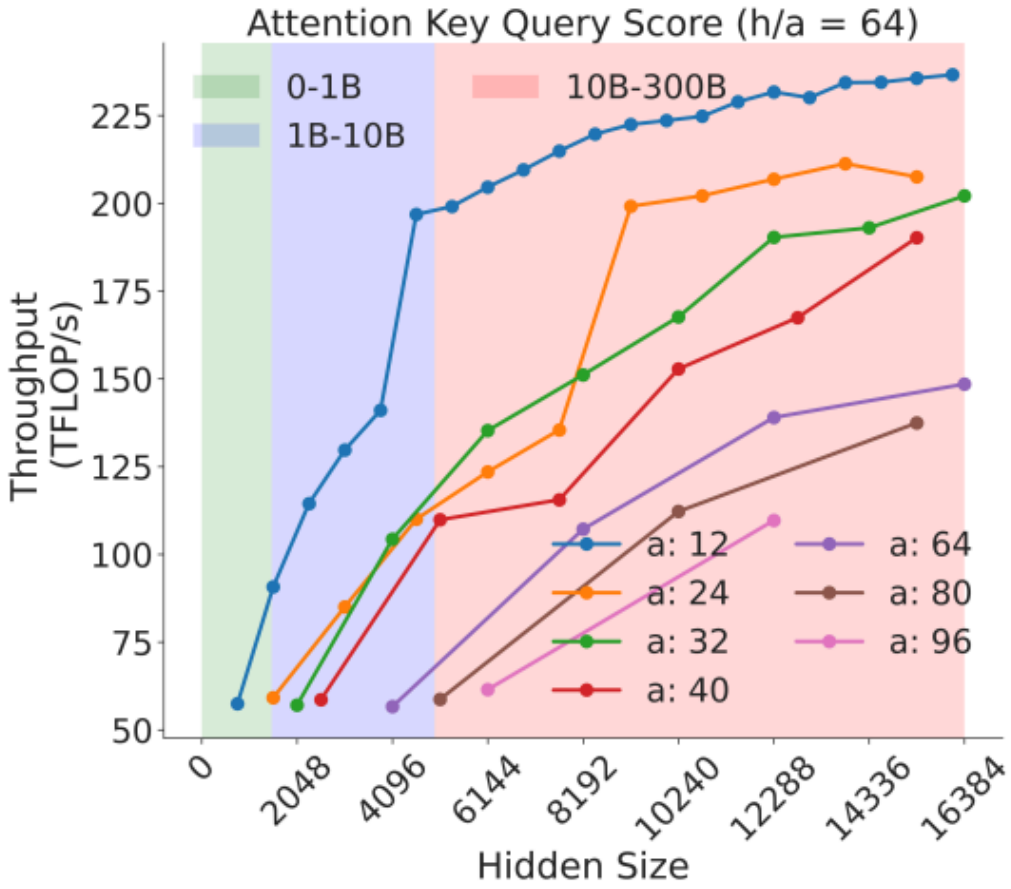


Fig. 8: Attention key-query score GEMM throughput assuming fixed ratio of $\frac{h}{a} = 64$ on A100 GPU

Flash attention

If you're using Flash Attention, good news! These MHA sizing constraints are taken care of for you. Your only constraint is to have a large enough ratio of `h/a` to saturate your GPU cores:

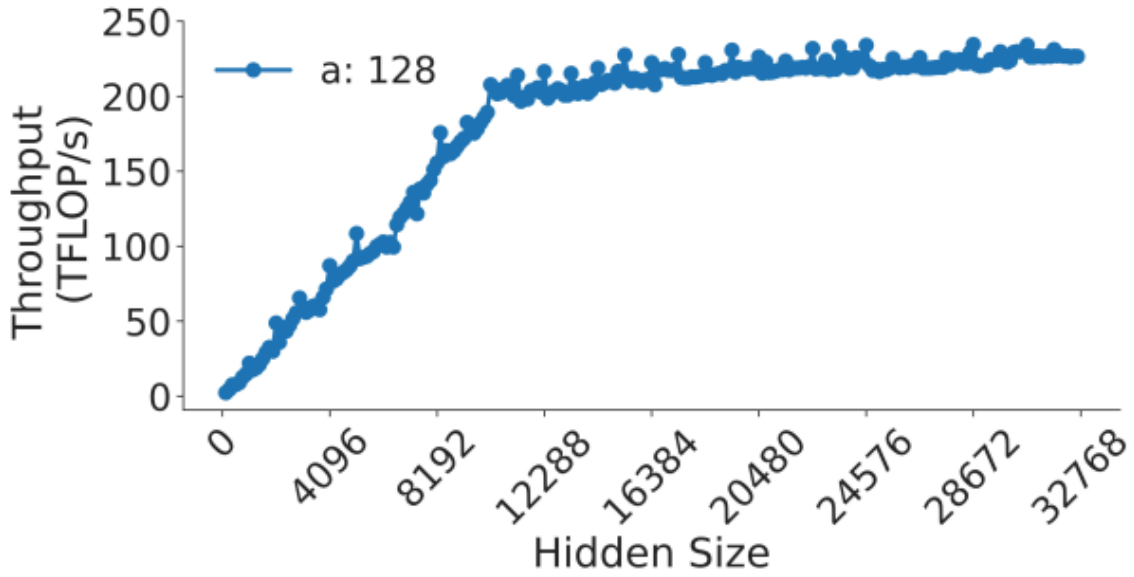


Fig. 12: Sweep over hidden dimension for FlashAttention (v2) [8] on NVIDIA A100 GPU.

Which *version* you run matters as much as whether you run it at all, because each major version targets an accelerator generation - FA3 for Hopper, FA4 for Blackwell - and an older version on newer hardware leaves the new architecture's features unused. This book measures the difference: on an 8B causal attention shape, FA4 on B200 against FA3 on H200 gives **1.88x on `forward+backward` at 8K sequence length, rising to 1.95x at 32K**, against a 2.28x hardware ratio - so attention is close to the ceiling and is *not* what limits that particular upgrade. See FA3 vs FA4 for the full comparison and the scripts that produce it.

The trap is the gap between silicon and kernels. A fully usable FA4 took many months to appear - it was still beta when the measurement above was taken - and until then the fallback was FA2, which is very slow on Blackwell precisely because it cannot exploit the new architecture. So a new accelerator can be sitting in the rack while its attention kernel is not ready, which makes the version you can actually run a constraint on the upgrade rather than a detail of it; Software support: the cost of switching too early treats this as a hard gate. HF Transformers auto-selects the backend per accelerator through `attn_implementation`, so check which version it actually chose instead of assuming the newest.

SwiGLU-based MLP

Models such as PaLM, LLaMA, Mistral and others use the SwiGLU activation function in place of the more common GLU activation function.

The SwiGLU-based MLP contains an additional learned matrix in its activation function. There the MLP block contains 3 matrices instead of the original 2. To preserve the total number of parameters in the MLP block the paper that introduces SwiGLU proposes to use `dim_mlp = 8/3*dim_attn` instead of the typical `dim_mlp = 4*dim_attn`. The The Case for Co-Designing Model Architectures with Hardware paper provides recommendations for finding the value of hidden dimension (`h`) that would lead to the best `matmul` performance, and if you used `8/3*h` is likely to result in a much slower MLP block, because `8/3` will break all the alignments.

In order to overcome this problem one only needs to realize that the `8/3` coefficient is only a suggestion and thus it's possible to find other near by coefficients that would lead to better-shaped MLP matrices. In fact if you look at the publicly available LLama-2 models, its 7B variant uses `11008/4096 = 2.6875` as a coefficient, which is quite close to `8/3 = 2.667`, and its 70B variant uses a much larger `28672/8192 = 3.5` coefficient. Here the 70B variant ended up with an MLP block

that contains significantly more parameters than a typical transformer block that doesn't use SwiGLU.

Now that we know the recommended coefficient isn't exact and since a good `h` has already been chosen, one can now search for a good nearby coefficient that still leads to high-performance GEMMs in the MLP. Running a brute-force search reveals that Llama-2-7B's intermediate size is indeed one of the best performing sizes in its range.

Here is swiglu-maf-bench.py that can be easily adapted to your use-case and once run on the target hardware the training will happen on, you will be able to find the best hidden size of the MLP.

Let's run it on H100 with `h = 4096`:

```
./swiglu-maf-bench.py
Wanted the closest to 10922 d_ff value that leads to the highest TFLOPS (d_hidden=4096)

Searching 50 steps in the range of 10822 .. 11022
Results: baseline, followed by near-by best performing d_ff results:

 d_ff  tflops mlp_params
------------------------
10922  272.73 134209536
------------------------
10944  398.38 134479872
10848  395.62 133300224
10880  395.52 133693440
10912  395.16 134086656
11008  395.01 135266304
```

As it can be easily seen the `8/3*4096=10922` leads to a rather slow performance. But `10944`, which is just `22` bigger than `10922`, gives a whooping `46%` speedup for the `matmul`. The total corresponding MLP parameters is printed as well, should you want to choose slightly slower choices but with a different number of parameters.

Final recommendations for model sizing

The full recommendations are:

1. Vocab size divisible by `64`
2. Microbatch size as large as possible
3. `b*s`, `h/a`, and `h/t` should be divisible by a power of 2
4. `(b*a)/t` should be an integer
5. `t` should be small as possible
6. For SwiGLU search for the best performing hidden size close to `8/3*h`

Additional reading

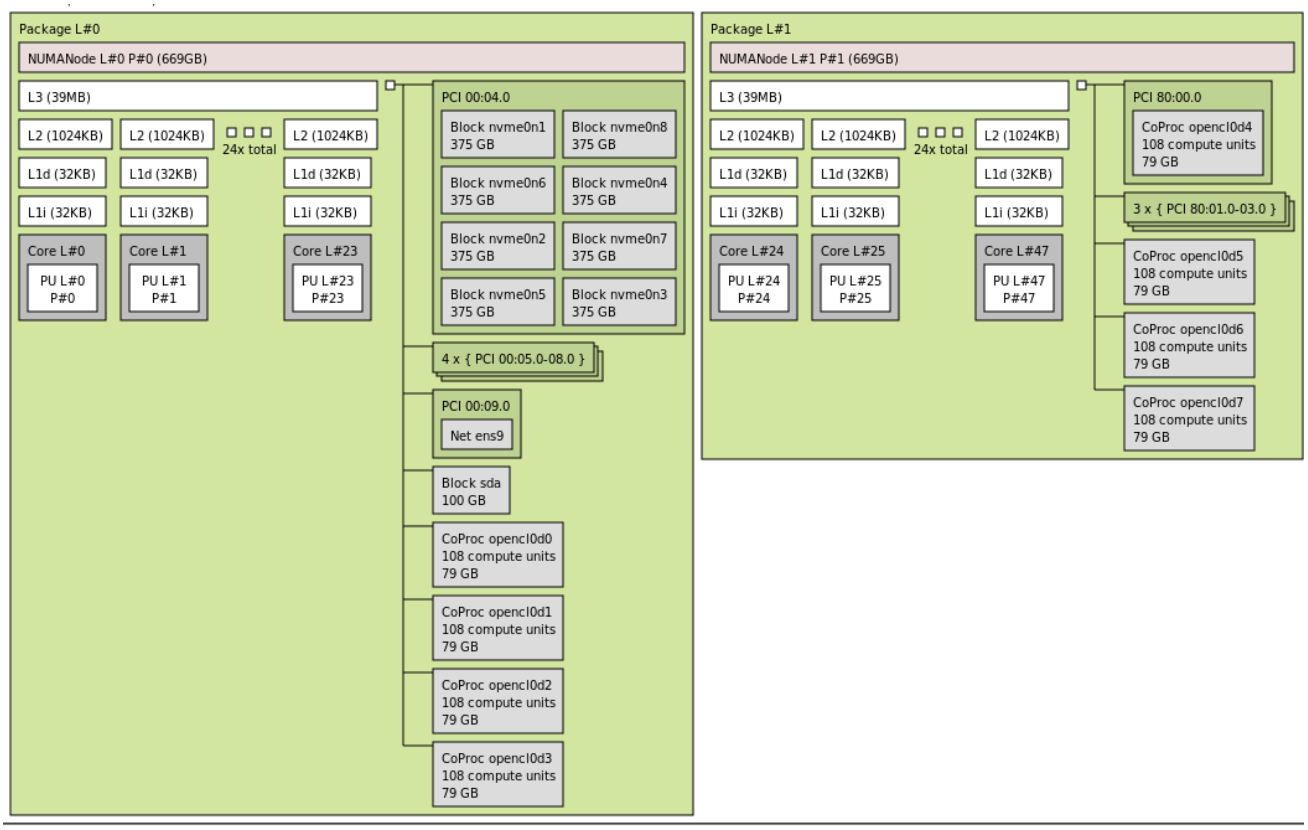
- What Shapes Do Matrix Multiplications Like?

NUMA affinity

Non-uniform memory access (NUMA) is a computer memory design used in multiprocessing, where the memory access time depends on the memory location relative to the processor. As modern servers have more than one CPU to get the best performance accelerators residing in the same NUMA node as the corresponding CPU should have the processes bound to that same NUMA node.

First, let's understand what do NUMA nodes signify.

Here is a typical A100 8x GPUs server NUMA nodes diagram:



As you can see it has 2 CPUs, each defining a NUMA block, and each such block contains a group of 4 GPUs. The GPUs are the gray blocks that say `CoProc` with 108 compute units (SMs) and 79GiB of memory.

footnote: the diagram was generated by `lstopo a100.png` from hwloc.

If you're using Hyper-Threads then you want to use `lstopo -l` to see the HT core count presented correctly. For example if you have 2 NUMA nodes with 8 accelerators and 104 physical cpu-cores and 208 logical cores - thus (`208/8=26` HT-cores per GPU), then the HT cores will be for:

- gpu0..3 `[0, 1, 2, 3, ..., 51, 104, 105, 106, ..., 129]`
- gpu4..7 `[52, 53, 54, ..., 103, 156, 157, 158, ..., 207]`

You first get the physical cpu-core counts and then the remaining HT cores, hence the strange gap.

If this is an NVIDIA node, the other way to easily see the CPU affinity is to run:

```
$ nvidia-smi topo -m
        GPU0    GPU1    GPU2    GPU3    GPU4    GPU5    GPU6    GPU7    CPU Affinity    NUMA
Affinity
GPU0     X      NV18    NV18    NV18    NV18    NV18    NV18    NV18    0-51,104-155    0
GPU1    NV18     X      NV18    NV18    NV18    NV18    NV18    NV18    0-51,104-155    0
GPU2    NV18    NV18     X      NV18    NV18    NV18    NV18    NV18    0-51,104-155    0
GPU3    NV18    NV18    NV18     X      NV18    NV18    NV18    NV18    0-51,104-155    0
GPU4    NV18    NV18    NV18    NV18     X      NV18    NV18    NV18    52-103,156-207  1
GPU5    NV18    NV18    NV18    NV18    NV18     X      NV18    NV18    52-103,156-207  1
GPU6    NV18    NV18    NV18    NV18    NV18    NV18     X      NV18    52-103,156-207  1
```

```
GPU7    NV18    NV18    NV18    NV18    NV18    NV18    NV18     X      52-103,156-207  1
```

On this 8-gpu node you can see the `CPU Affinity` column which tells you which cores reside together with the first and the second group of GPUs, and the `NUMA Affinity` column.

Now that it's clear that the various compute components are placed in 2 or more groups, to achieve the best performance we need to ensure that the components communicate within the group they belong to, and avoid any cross-talk. For example, if gpu0 belong to NUMA node 0, then the process that drives this GPU should only use cpu-cores from NUMA node 0.

The same should apply to networking or any other components that you may have control over.

Practically though in my experience so far if your workload is very light on CPU work this change will make very little difference to the overall performance, but can be quite impactful if a lot of CPU use is done. On the other hand if doing the most efficient thing is easy, even the tiniest improvement is likely to accumulate over long training jobs, so it's worth to implement, IMHO.

NUMA process binding

There are multiple ways to accomplish the binding of processes to the cpu-cores of the right NUMA node.

numactl

One of the most common tools to do that is using `numactl`, which sets the NUMA affinity as it launches a new process.

For example, let's see how it can be integrated with the `torchrun` launcher.

This launcher currently needs a helper util [numa-set.sh] to perform NUMA affinity settings, once you downloaded it and made it executable, you can now get the right NUMA affinity using:

```
torchrun --nproc_per_node=8 --role : --tee 3 --no-python ./numa-set.sh your-program.py
```

Note: you'd need `numactl` installed on your system for this util to work.

For example, here is how you can validate that the assignments are correct:

```
torchrun --nproc_per_node=8 --role : --tee 3 --no-python ./numa-set.sh python -c \
'import os; cores=os.sched_getaffinity(0); print(f"{len(cores)} visible cpu cores: {cores}")'
```

On a system with 208 HT cpu-cores, you will most likely see:

```
[:0]:104 visible cpu cores: {0, 1, 2, 3, 4, 5...
[:1]:104 visible cpu cores: {0, 1, 2, 3, 4, 5...
[:2]:104 visible cpu cores: {0, 1, 2, 3, 4, 5...
[:3]:104 visible cpu cores: {0, 1, 2, 3, 4, 5...
[:4]:104 visible cpu cores: {52, 53, 54, 55, ...
[:5]:104 visible cpu cores: {52, 53, 54, 55, ...
[:6]:104 visible cpu cores: {52, 53, 54, 55, ...
[:7]:104 visible cpu cores: {52, 53, 54, 55, ...
```

The first 4 accelerators use the first half of the cpu-cores and the other 4 the second half, which matches the earlier explanations of the right setting.

If you remove `./numa-set.sh`, you'd get:

```
torchrun --nproc_per_node=8 --role : --tee 3 --no-python python -c \
'import os; cores=os.sched_getaffinity(0); print(f"{len(cores)} visible cpu cores: {cores}")'
```

You will see that all 8 processes see all 208 cpu-cores:

```
[:0]:208 visible cpu cores: {0, 1, 2, 3, ...
```

so as each process has access to any cpu-core - a cross talk may occur, which may introduce a small performance overhead.

os.sched_setaffinity

You can, of course, change the NUMA affinity after the program was launched. You saw the use of `os.sched_getaffinity` to get the current settings, and the corresponding `os.sched_setaffinity` is used to change it.

```
import os
os.sched_setaffinity(0, [0, 1])
```

Here we told the system that the process running this script (`0`) can only use cpu-cores `0` and `1`.

So now we just need to figure out how to programmatically get the right cpu sets for each accelerator's process. Here is how to do it with pynvml.

pynvml

If you're using NVIDIA GPUs, `pynvml` (`pip install nvidia-ml-py` - the module is still imported as `pynvml`) can be very helpful to get all sorts of information about the gpu and not needing to call `nvidia-smi` - in this situation we are going to use it to tell us the correct CPU affinity given a GPU index.

In numa-set-pynvml.py you will find a working helper function that you could call at the very top of your training loop like so:

```
import os
gpu_index = int(os.environ.get("LOCAL_RANK", 0))
set_numa_affinity(gpu_index, verbose=True)
```

call it before `DataLoader` is initialized to get the workers use the right cpu-cores!

Normally, the local process rank equals the gpu index, but if one uses `CUDA_VISIBLE_DEVICES` - this might not be true any longer - if you use it, you will need to remap the process rank to the actual index:

```
gpu_index = int(os.environ.get("LOCAL_RANK", 0))
if "CUDA_VISIBLE_DEVICES" in os.environ:
    ids = list(map(int, os.environ.get("CUDA_VISIBLE_DEVICES", "").split(",")))
```

```
    gpu_index = ids[gpu_index] # remap
```

The other gotcha can be `CUDA_DEVICE_ORDER` which typically defaults to `PCI_BUS_ID`, but one could also set it to `CUDA_DEVICE_ORDER=FASTEST_FIRST` if you have mixed GPUs, but it's very very unlikely that you will run into this in a high end server setup, so you can safely ignore this.

srun

If using SLURM and you're OK with using `srun` as the launcher, rather than `torchrun`, `accelerate`, etc., it'll do all the binding work for you automatically. See the full launcher here.

To make it NUMA affinity-ready all you need to add is these 2 headers:

```
#SBATCH --gres-flags=enforce-binding
#SBATCH --ntasks-per-socket=4
```

`--ntasks-per-socket=4` assumes you have 2 cpu sockets with 8 accelerators - so `8/2=4` accelerators per socket.

This is an even more precise solution, since it'd assign each process its own group of cpu-cores, rather than just give all the NUMA node 0 cpu-cores to the processes driving accelerators 0-3, and NUMA node 1 cpu-cores to the processes driving accelerators 4-7.

Specific launchers

Various launchers have support for NUMA affinity settings:

- HF Accelerate has a flag `--enable_cpu_affinity` that you add to the `accelerate` launch command and it'll do this for you. Available since `accelerate>0.28.0`.
- torchrun doesn't have it, but I showed how to do it in this section.
- srun was covered here.

DataLoader

Asynchronous DataLoader

The default setting is `num_workers=0`, which means whenever you call `next(iter(dataloader))` the data is actively fetched in real time - which means there will be an IO overhead and if there are any transforms those would be applied in real time as well - all blocking the accelerator's compute.

The solution is to use the asynchronous `DataLoader` by setting `num_workers > 0`. Typically, unless your `DataLoader` is extremely slow 2 workers is enough:

```
DataLoader(..., num_workers=2, ...
```

Now when `next(iter(dataloader))` is called the data should be already in the CPU memory with all the transforms done. It still needs to be copied to the accelerator memory - to speed that up see Pinned memory and non blocking device copy.

Here is a benchmark which emulates a slow data transform: num-workers-bench.py

```
num_workers=0: average time: 5.388
```

```
num_workers=1: average time: 3.900
num_workers=2: average time: 1.333
num_workers=3: average time: 0.839
num_workers=4: average time: 0.875
```

So you can see that in this particular case the speed was dramatically improving up to 3 workers. If the `DataLoader` is very light and does very little the difference will be much smaller, but 0 workers will always lead to the biggest overhead.

By measuring the performance of your workload you can finetune this number by trying lower and higher values. But remember that each one of the workers may consume a lot of CPU memory. So on a node of 8 accelerators with 2 workers, that would be 16 additional processes. Nowadays, the compute nodes have often hundreds of cpu cores and TBs of CPU memory so there should be plenty of resources for many workers to be supported. In the past it was a different story.

Also note that because any data transforms are applied asynchronously and ahead of time, the CPU and memory speed don't matter much in this case. e.g. with 2 workers as long as the next iteration data preparation takes less than 2 compute iterations the `DataLoader` shouldn't be a bottleneck.

Beware that sometimes you may encounter problems using `num_workers > 0` - the pytorch Issues has a few related Issues that haven't been resolved in several years, where a worker would hang. In particular when having 2 `Dataloaders`. In fact we had this problem during BLOOM-176B training, where the training `Dataloader` worked fine with 2 workers, but once eval `Dataloader` was added it'd randomly hang - so we after failing to figure it out we resorted to a workaround of `num_workers=0` just for the eval and switch to doing it very rarely. Eventually, we stopped doing eval altogether and started doing lm-harness style async eval done to the saved intermediary checkpoints instead, which also sped up the training process.

case study: during IDEFICS-80B training we were using a streaming `Dataloader` which worked really bad and it was consuming huge amounts of memory per worker, and it'd spike often, and we had about 1TiB of CPU memory and we couldn't spawn enough workers - so the `Dataloader` was a bottleneck. We didn't have time to find a better solution at that time so we finished training with it.

Pinned memory and non-blocking device copy

A combination of:

1. `DataLoader(pin_memory=True, ...)`
2. `batch.to(device="cuda", non_blocking=True)`

is likely to make the `DataLoader` less of a bottleneck.

1. Enabling pinned memory allows for a more efficient data transfer from CPU to accelerator memory.
2. non-blocking will further speed things up by allowing some overlap between compute and data movements

Here is a small benchmark demonstrating the difference: pin-memory-non-block-bench.py. When I run it on an A100 80GB-PCIe, the output was:

```
pin_memory= True, non_blocking= True: average time: 0.459
pin_memory= True, non_blocking=False: average time: 0.522
pin_memory=False, non_blocking= True: average time: 0.658
pin_memory=False, non_blocking=False: average time: 0.646
```

so you can see that `pin_memory=True+non_blocking=True` is a worthy change.

For more background you can read 1 and 2.

Notes:

- Pinned memory is treated specially by the OS, by preventing the paging out when the total available memory is low, so it reduces the total amount of available memory to other programs. Thus use with care.
- I recall that a few times people reported problems when using pinned memory - I think it's mostly when the system doesn't have much CPU memory to start with or they used too much of pinned memory, so the OS can start swapping heavily.
- If you measure your per iteration TFLOPS you can compare the throughput with and w/o these changes and choose the one that works the best. It'd be even easier to see the impact if you measure the `DataLoader` overhead separately from the `forward/backwards` and post-compute (usually logging, which can be surprisingly slow at times).

torch.compile

`torch.compile` speeds up both training and inference by tracing your model into a graph and generating fused kernels for it. It shipped in PyTorch 2.0 and as of 2026-08 much of the ecosystem runs it by default, but getting a good speedup on an arbitrary model is still not automatic - the tracer has to be able to see your code, and that is where the work is.

Where it pays off

Look back at the three groups in Anatomy of Model's Operations, because they behave very differently under compilation:

1. **Tensor contractions** already dispatch to vendor GEMM libraries running near the hardware limit, so there is little here for a compiler to win. `mode="max-autotune"` will search Triton matmul variants and can occasionally beat the vendor library, but this is the least promising group.
2. **Statistical normalizations** and 3. **element-wise operators** are where the wins are. These are memory-bandwidth-bound rather than compute-bound: each reads its input from HBM, does very little arithmetic, and writes the result back. Eager mode pays that round-trip once per operation. Fusing a chain of them into a single kernel pays it once for the whole chain. That is the bulk of what `torch.compile` does to a transformer, and it is why a model made of many small operations gains more than one dominated by large matmuls.

So the speedup to expect is a function of how much of your step time sits in group 2 and 3 work. Profile first with the PyTorch performance profilers, then compile.

A separate win applies when you are launch-bound rather than bandwidth-bound: `mode="reduce-overhead"` captures the step into CUDA graphs and replays one launch sequence instead of issuing thousands of individual kernel launches. This matters most at small batch sizes and in low-latency inference decode, where the accelerator sits idle waiting for the CPU to feed it. It costs memory, since the captured graph pins its buffers, and it needs static shapes.

Two measurements on a single NVIDIA B200 with `torch=2.13.0+cu130`, taken 2026-08-09, bracket the range; in both, the `forward` row is inference under `no_grad` while the two backward rows enable gradient checkpointing, which is standard practice at this size and is what keeps the 8B step within one GPU's memory. The first is a small model at a short sequence - Llama 3.2 1B (`NousResearch/Llama-3.2-1B`), bf16, SDPA, batch `1` x sequence `512` - where each step is a handful of milliseconds and is therefore dominated by kernel-launch overhead and the memory-bandwidth-bound work of groups 2 and 3. This is `torch.compile` at its most flattering. Steady-state step time after warmup, ratio against eager in parentheses:

workload	eager	`torch.compile()`	`mode="reduce-overhead"`	`mode="max-autotune"`
forward (infer)	8.1ms	3.4ms (2.4x)	1.9ms (4.2x)	2.0ms (4.1x)
forward+backward	48.8ms	12.6ms (3.9x)	8.1ms (6.0x)	8.0ms (6.1x)
+ AdamW step	58.8ms	18.7ms (3.1x)	17.6ms (3.3x)	17.5ms (3.4x)

The second is a large model at a long sequence - Llama 3.1 8B (`meta-llama/Llama-3.1-8B`), same dtype and attention backend, batch 1 x sequence 8192 - where each step is hundreds of milliseconds and almost all of that time sits in large matmuls (group 1) that already run near the hardware limit on the vendor GEMM library:

workload	eager	`torch.compile()`	`mode="reduce-overhead"`	`mode="max-autotune"`
forward (infer)	152.4ms	155.6ms (0.98x)	155.8ms (0.98x)	154.8ms (0.98x)
forward+backward	590.7ms	630.2ms (0.94x)	629.2ms (0.94x)	633.7ms (0.93x)
+ AdamW step	637.3ms	673.8ms (0.95x)	671.4ms (0.95x)	680.0ms (0.94x)

The gap between the two tables is the whole lesson. The small overhead-bound model gains up to about 6x; the large compute-bound one gains nothing - every compiled mode lands within a few percent of eager, and here on the slow side of it. Compilation did not fail; there was simply nothing for it to remove. Two effects stack on the 1B model. First, its plain step is launch- and bandwidth-bound, so fusing the many small kernels is a large win by itself. Second, the two backward rows enable gradient checkpointing, which discards activations and recomputes the whole forward during the backward - so each of those steps runs the forward twice. On a tiny model that extra forward is itself launch-bound, and compile fuses the recomputed kernels just as cheaply as the originals, absorbing most of it - which is why the `forward+backward` and `+ AdamW` rows show a larger speedup than the `forward`-only row.

On the 8B step at this sequence length neither effect exists: the GEMMs already run at cuBLAS peak and dominate, the checkpointing recompute is just more of those same peak-bound matmuls, and the launch gaps compile would close are a rounding error against half-second kernels. That is why all three compiled modes collapse onto the same ~0.94x - `max-autotune`'s Triton matmul search cannot beat cuBLAS on these shapes, and `reduce-overhead`'s CUDA graphs remove a launch overhead that is already negligible. So the speedup to expect tracks how bandwidth- or launch-bound your step is, which is exactly what the profiling in Where it pays off is meant to tell you: a model of many small operations, or any model run at a small batch and short sequence, gains the most, while a dense-matmul workload at a large shape gains the least - sometimes less than nothing.

The cold-start cost is real and, when the steady-state gain is small or absent, is what decides whether compiling is worth it at all. With a fresh Inductor cache the first `forward+backward` step took about 22s (1B) and 47s (8B) under default compile, rising to 83s (1B) and 168s (8B) under `max-autotune`. On the 1B model that buys back quickly: default compile saves ~35ms per step, so it breaks even against eager in roughly 600 steps and everything after is profit. On the 8B/8192 case there is no steady-state saving to amortize - the compiled step is a hair slower - so the compile time is pure cost and never pays back on this shape, the sharpest possible version of "fatal for a ten-step smoke test." The optimizer step does not add to the compile bill: compiling the model is the cost, and once the forward/backward graphs exist, folding `opt.step()` into the timed region does not recompile - note the near-zero cold column on the `+ AdamW step` rows, where the graph was already built by the `forward+backward` case earlier in the same run. Compilation also improves between PyTorch releases, so treat the absolute milliseconds - and the sign of the 8B result - as dated to this toolchain rather than as a permanent property of the models.

If you want to experiment yourself, both tables come from torch-compile-bench.py: a single loop over the workloads and modes that builds a fresh model per cell, enables gradient checkpointing on the backward workloads (which also keeps the 8B/8192 step within one GPU's memory), times the first step as the cold-start cost and the mean of the following steps as the steady state, and prints one row per mode with the ratio against eager. Model, sequence length, batch, seed, and the mode list are variables at the top of the file (`SEED = 42`); the 8B figures above are the defaults, and the 1B table comes from uncommenting the two `NousResearch/Llama-3.2-1B` / `SEQ = 512` lines (batch stays 1). The seed pins the random input tokens so re-runs see the same data.

What makes it hard

The tracer has to turn your `forward` into a graph, and anything it cannot trace becomes a **graph break** - it compiles what it

can, falls back to eager for the untraceable part, then starts a new graph. Every break is also a fusion boundary, so a model with many breaks can "work" under `torch.compile` and gain almost nothing. Data-dependent control flow, `.item()`, `print` and unsupported operations all break the graph.

The other trap is **recompilation**. A compiled graph is specialized to the shapes it was traced on, so a new sequence length triggers a recompile, and that only happens a bounded number of times (`torch._dynamo.config.recompile_limit`, 8 by default) before the compiler gives up on that function and runs it eager from then on. Variable-length batches are the usual cause, and they are common - so if your inputs vary in shape, `torch.compile(model, dynamic=True)` is the normal answer. It compiles one shape-agnostic graph up front, giving up some peak performance in exchange for never recompiling. Reach for it as the default when shapes vary, rather than as a fix after you notice the recompiles.

Neither trap is mysterious - both are printable:

```
TORCH_LOGS="graph_breaks,recompiles" python train.py
```

which reports each break against the line of your code that caused it, and each recompile against the guard that failed. `torch.compile(model, fullgraph=True)` promotes breaks to errors, which is the quickest way to find all of them at once.

Compilation itself costs wall-clock time on the first step, and again after any change that invalidates the cache. When the model is one block repeated `n` times, compiling the block instead of the whole model cuts that cost sharply, because the compiler does the work once and reuses it for every layer.

If you tried it and things don't work you:

1. may report it to the PyTorch team, ideally with a small reproducible example
2. can work through PyTorch's own torch.compile troubleshooting guide, which is maintained alongside the compiler and has sections for each of the failure modes above, and you might be able to make some things work, and may still need to report some issues to PyTorch

Use a recent PyTorch release - the compiler improves substantially between versions, and a problem you hit on an older one may not exist on the current stable.

Automatic garbage collection

Python periodically performs an automatic garbage collection based on internal heuristics. In an LLM-training scenario with hundreds to thousands of accelerators used in synchronization - if different ranks follow even slightly different code paths the automatic garbage collection process could be triggered at different times for different ranks. Which means that one or more ranks could be slower than other ranks while performing this operation, and thus becoming stragglers, slowing down the whole ensemble.

Usually one can see this by studying the MFU plot where downward spikes can be observed.

If this happens to your training you can disable the automatic garbage collection with:

```
import gc
gc.disable()
```

at the beginning of your trainer and then manually perform garbage collection at the desired intervals. For example, calling this once in a training iteration:

```
import gc
```

```
gc.collect()
```

Refer to gc's manpage for more nuances.

Memory profiler tools

In this chapter we discussed the theoretical math of how much this or that feature should consume in MBs of memory. But often in reality things aren't exactly the same. So you plan for a certain model size and batch sizes but when you come to use it suddenly there is not enough memory. So you need to work with your actual code and model and see which part takes how much memory and where things got either miscalculated or some additional missed overhead hasn't been accounted for.

You'd want to use some sort of memory profiler for that purpose. There are various memory profilers out there.

One useful tool that I developed for quick and easy profiling of each line or block of code is IPyExperiments. You just need to load your code into a jupyter notebook and it'll automatically tell you how much CPU/GPU memory each block allocates/frees. So e.g. if you want to see how much memory loading a model took, and then how much extra memory a single inference step took - including peak memory reporting.

Fault Tolerance

Regardless of whether you own the ML training hardware or renting it by the hour, in this ever speeding up domain of ML, finishing the training in a timely matter is important. As such if while you were asleep one of the GPUs failed or the checkpoint storage run out of space which led to your training crashing, you'd have discovered upon waking that many training hours were lost.

Due the prohibitively high cost of ML hardware, it'd be very difficult to provide redundancy fail-over solutions as it's done in Web-services. Nevertheless making your training fault-tolerant is achievable with just a few simple recipes.

As most serious training jobs are performed in a SLURM environment, it'll be mentioned a lot, but most of this chapter's insights are applicable to any other training environments.

Always plan to have more nodes than needed

The reality of the GPU devices is that they tend to fail. Sometimes they just overheat and shut down, but can recover, at other times they just break and require a replacement.

The situation tends to ameliorate as you use the same nodes for some weeks/months as the bad apples get gradually replaced, but if you are lucky to get a new shipment of GPUs and especially the early GPUs when the technology has just come out, expect a sizeable proportion of those to fail.

Therefore, if you need 64 nodes to do your training, make sure that you have a few spare nodes and study how quickly you can replace failing nodes should the spares that you have not be enough.

It's hard to predict what the exact redundancy percentage should be, but 5-10% shouldn't be unreasonable. The more you're in a crunch to complete the training on time, the higher the safety margin should be.

Once you have the spare nodes available, validate that your SLURM environment will automatically remove any problematic nodes from the pool of available nodes so that it can automatically replace the bad nodes with the good ones.

If you use a non-SLURM scheduler validate that it too can do unmanned bad node replacements.

You also need at least one additional node for running various preventative watchdogs (discussed later in this chapter), possibly offloading the checkpoints and doing cleanup jobs.

Rebuilding the process group in place

The rest of this chapter recovers from a dead rank by exiting and letting the next queued job start on a fresh allocation. Starting from PyTorch 2.14 the communicator itself can be rebuilt in place, so surviving ranks keep their warm process state. That path is experimental as of 2.14: you opt in at `init_process_group(..., enable_reconfigure=True)`, and the calls (`dist._get_reconfigure_handle`, `dist._reconfigure`) are underscored and documented as subject to change. Until they stabilize, keep using spare nodes and a job array as the recovery path.

Queue up multiple training jobs

The next crucial step is to ensure that if the training crashed, there is a new job lined up to take place of the previous one.

Therefore, when a training is started, instead of using:

```
sbatch train.slurm
```

You'd want to replace that with:

```
sbatch --array=1-10%1 train.slurm
```

This tells SLURM to book a job array of 10 jobs, and if one of the job completes normally or it crashes, it'll immediately schedule the next one.

footnote: `%1` in `--array=1-10%1` tells SLURM to launch the job array serially - one job at a time.

If you have already started a training without this provision, it's easy to fix without aborting the current job by using the `--dependency` argument:

```
sbatch --array=1-10%1 --dependency=CURRENTLY_RUNNING_JOB_ID train.slurm
```

So if your launched job looked like this:

```
$ squeue -u `whoami` -o "%.10i %9P %20j %.8T %.10M %.8l %.6D %.20S %R"
     JOBID PARTITION NAME             STATE       TIME   TIME_LIM    NODES  START_TIME
NODELIST(REASON)
       87    prod    my-training-10b  RUNNING 2-15:52:19 1-16:00:00   64    2023-10-07T01:26:28
node-[1-64]
```

You will note that the current's `JOBID=87` and now you can use it in:

```
sbatch --array=1-10%1 --dependency=87 train.slurm
```

and then the new status will appear as:

```
$ squeue -u `whoami` -o "%.10i %9P %20j %.8T %.10M %.8l %.6D %.20S %R"
     JOBID PARTITION NAME             STATE       TIME   TIME_LIM    NODES  START_TIME
NODELIST(REASON)
       87    prod    my-training-10b  RUNNING 2-15:52:19 1-16:00:00   64    2023-10-07T01:26:28
node-[1-64]
 88_[1-10%1]   prod    my-training-10b  PENDING       0:00 1-16:00:00   64                   N/A
(Dependency)
```

So you can see that an array of 10 jobs (`88_[1-10%1]`) was appended to be started immediately after the current job (`87`) completes or fails.

Granted that if the condition that lead to the crash is still there the subsequent job will fail as well. For example, if the storage device is full, no amount of restarts will allow the training to proceed. And we will discuss shortly how to avoid this situation.

But since the main reason for training crashes is failing GPUs, ensuring that faulty nodes are automatically removed and the new job starts with a new set of nodes makes for a smooth recovery from the crash.

In the SLURM lingo, the removed nodes are given a new status called `drained`. Here is an example of a hypothetical SLURM cluster:

```
$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
prod*       up   infinite       4  drain node-[0-3]
prod*       up   infinite      47  alloc node-[4-51]
prod*       up   infinite      23   idle node-[52-73]
```

Here we have 47 nodes being used (`alloc`), 23 available (`idle`) and 4 unavailable (`drained`).

The sysadmin is expected to periodically check the drained nodes, fix or replace them and then make them again available to be used by changing their state to `idle`.

The other approach is to daisy-chain jobs via `--dependency` as explained here. Both of these approaches could also be combined.

How do you know when the job array or a daisy chain should not resume - well, normally the training loop will exit immediately if it knows the job is done. But you could also add features like kill switch which are even easier to use to prevent a job array from running.

Preferring fixed accelerator allocations to dynamic ones

Typically when getting a new set of accelerator nodes, especially when it's a new type of an accelerator that came out recently, many accelerators will fail, making LLM training quite problematic. There can be as large as 10% failure rate early on for new accelerators and still pretty high percentage of failures at later stages. Remember that if you have 8 accelerators, even one failing accelerator is like all 8 failing from the perspective of the training program.

If you use a fixed node allocation, after a few months, the bad accelerators will be weeded out and there should be very few accelerators failing. It'll still happen but it'll be a rare event.

Make sure your provider gives you new accelerators when they fail and doesn't return you the same accelerators after having them cool off (literally). For example, see how to track the NVIDIA GPUs UUID. Those transient failures are likely to repeat when under heavy stress, so you want those to be replaced for real.

If you use a dynamic allocation, even a year after a new accelerator type has been released, expect lots of failing accelerators, since you'd be getting rejected nodes by other users. Surely, some clouds are better than others at diligently replacing bad hardware, the problem is that there are many accelerators that don't fail outright and when someone dropped a bad node, the technician looking at it may not see any problems with it when they try it out. And if the user just released the node without reporting it was broken, if the cloud provider isn't re-checking that a node is kosher before giving it to the next user the probability of getting a bad node is extremely high.

Frequent checkpoint saving

Whenever the training job fails, many hours of training can be lost. This problem is mitigated by a frequent checkpoint saving. When the training is resumed it'll continue from the last checkpoint saved. If the failure occurred 12 hours after the last checkpoint has been saved, 12 hours of training is lost and needs to be re-done. This can be very expensive if the training uses hundreds of GPUs.

In theory one could save a checkpoint every 10 minutes and only ever lose 10 minutes of training time, but this too would dramatically delay the reaching of the finish line because large models can't be saved quickly and if the saving time starts to create a bottleneck for the training this approach becomes counterproductive.

Depending on your checkpointing methodology and the speed of your IO storage partition the saving of a large model can take from dozens of seconds to several minutes. Therefore, the optimal approach to saving frequency lies somewhere in the middle.

The math is quite simple - measure the amount of time it takes to save the checkpoint, multiply it by how many times you'd want to save it and see how much of an additional delay the checkpoint saving will contribute to the total training time.

Use case: While training BLOOM-176B we had an incredibly fast GPFS over NVME filesystem and it took only 40 seconds to save a 2.3TB checkpoint written concurrently on 384 processes. We saved a checkpoint approximately every 3 hours. As we trained for about 3 months, that means that we saved about 720 checkpoints (`90 days * 24h / 3h`) - that is an additional 8 hours was spent just saving the checkpoints (`720 times * 40 secs / 3600 secs`) - or ~0.37% of the total training time (`8h / (90 days * 24 hours)`). Now say if the IO were to be 5 times slower, which is not uncommon on the cloud unless one pays for premium IO, that would have become 2% of the training time, which would be quite significant.

footnote: If you don't have a large local storage and you have to offload the checkpoints to the cloud, make sure that the 2 most frequent checkpoints remain local to allow for a quick resume. The reason for 2 and not 1, is that it's possible that the very last checkpoint got corrupted or didn't finish saving if a crash occurred during its saving.

While this method introduces an overhead to the training, having training checkpoints is hugely useful. Because these allow you to rollback many steps back should there be a divergence, are useful for analysis of various events and many trainings these day switch from in-training single loss measuring eval, which provide little useful signal to a full blown dataset-based evaluation on multiple benchmarks applied to each checkpoint during training. The latter can be done on additional nodes w/o slowing down the training for in-training evals.

Multi-Replica-based fault tolerance

There is another approach to dealing with accelerator crashes which involves no checkpoint saving. This approach only works in situations where at least two model replicas are used during training.

Please review the various model parallelism techniques first to be able to follow along.

- If some variation of 3D model parallelism is used, that is you have either Tensor Parallelism (TP) and/or Pipeline Parallelism (PP) and/or Data Parallelism (DP), the number of replicas is equal to the DP degree.
- If Hybrid ZeRO-DP parallelism is used, then the number of replicas is equal to the degree of hybrid replicas.

For example, say you have a training setup that uses a 3D parallelism of TP=4, PP=2, DP=2 - so then you have 2 replicas, each using 8 gpus `node0` and `node1` (TP=4, PP=2 => `4*2=8`) - practically, each replica uses a whole 8-GPU node.

Additionally you have a standby back up node `node2` with 8 GPUs idling but ready to be used at a moment's notice.

Now, say, during training `node0.gpu0` fails. Since you have a 2nd replica with intact data, you switch over to the standby 8GPU node, RDMA copy the data from the gpus of the 2nd replica and you can continue training where you left off. This is a very simplistic explanation since there are multiple nuances to figuring out such recovery depending at which stage of the iteration loop the failure occurred. It other words there is a complex algorithm to implement.

Of course, on a large scale training you're likely to have a hundred active nodes and a small handful of back up node.

This approach is superior to file system checkpointing saving because, you only ever lose one iteration, whereas with file system checkpointing you may lose hundreds of iterations.

The open-source library that lands closest to this idea is `torchft` (Meta / PyTorch). It treats each training step as a fault boundary across replica groups, coordinates health with a lighthouse/quorum service, and can heal a failed group by live state transfer from a healthy peer instead of rolling back to a filesystem checkpoint. Out of the box it covers fault-tolerant DDP and HSDP, and `torchtitan` wires it into an end-to-end HSDP training loop - see its `torchft` experiment docs and the PyTorch blog write-up of a checkpoint-free run under synthetic failures. The failure domain is a whole replica group rather than a single GPU, and the healing path is peer recovery through torchft's checkpoint transports rather than the

RDMA-copy-onto-a-dedicated-standby sketch above - same goal (lose at most a step, keep training), different machinery.

Related research systems attack the same problem from other angles. `Oobleck` (SOSP'23) keeps logically equivalent pipeline replicas and recovers by re-instantiating precomputed pipeline templates after failures, again without a full checkpoint restart. Earlier spot-instance work such as Bamboo (redundant computation scheduled into pipeline bubbles; NSDI'23) and Varuna (elastic PP+DP reconfiguration) is in the same family; those codebases are largely dormant now and are useful mainly as design references. Commercial infrastructure layers also exist that migrate a failed GPU onto a spare without changing the training script - a different trade-off (per-GPU failure domain, outside the framework) rather than another torchft clone.

Kill switch

In many SLURM environments users have no `sudo` access and when one user started a training and went to sleep, and then a problem has been discovered, the other users can't easily stop the training and restart it again.

This was the situation during BLOOM-176B training and we implemented a kill-switch to handle that. The mechanism is very simple. The training loop polls for a specific file to appear before starting a new iteration and if the file is there the program saves the checkpoint and exits, allowing users other than the one who started the previous training to change things and restart it again. An additional poll was added at the very beginning of `main` so that if there was a long job array queued by the user who is asleep they could be "burned through" quickly by getting each job exit quickly on start.

This is also discussed here.

This facility helps to minimize the amount of wasted training time.

Save switch

While mentioning the kill switch, it might be good to quickly mention its cousin, a save switch. Similarly to the kill switch the save switch is a variation of the former where instead of stopping the training, if the training loop discovers that a save-switch file appears - it will save a checkpoint, but will continue training. It'll also automatically remove the save-switch from the file-system, so that it won't accidentally start saving a checkpoint after every iteration.

This feature can be very useful for those who watch the training charts. If one sees an interesting or critical situation in the training loss or some other training metric one can quickly ask the training program to save the checkpoint of interest and be able to later reproduce the current situation at will.

The main use of this feature is around observing training loss spikes and divergences.

Prevention

The easiest way to avoid losing training time is to prevent certain types of problems from happening. While one can't prevent a GPU from failing, other than ensuring that adequate cooling is provided, one can certainly ensure that there is enough of disk space remaining for the next few days of training. This is typically done by running scheduled watchdogs that monitor various resources and alert the operator of possible problems long before they occur.

Scheduled Watchdogs

Before we discuss the various watchdogs it's critical that you have a mechanism that allows you to run scheduled jobs. In the Unix world this is implemented by the `crontab` facility.

Here is an example of how `~/bin/watch-fs.sh` can be launched every hour:

```
0 * * * * ~/bin/watch-fs.sh
```

The link above explains how to configure a crontab job to run at various other frequencies.

To setup a crontab, execute `crontab -e` and check which jobs are scheduled `crontab -l`.

The reason I don't go into many details is because many SLURM environments don't provide access to the `crontab` facility. And therefore one needs to use other approaches to scheduling jobs.

The section on Crontab Emulation discusses how to implement crontab-like SLURM emulation and also Self-perpetuating SLURM jobs.

Notification facility

Then you need to have one or more notification facilities.

The simplest one is to use email to send alerts. To make this one work you need to ensure that you have a way to send an email from the SLURM job. If it isn't already available you can request this feature from your sysadmin or alternatively you might be able to use an external SMTP server provider.

In addition to email you could probably also setup other notifications, such as SMS alerting and/or if you use Slack to send slack-notifications to a channel of your choice.

Once you understand how to schedule watchdogs and you have a notification facility working let's next discuss the critical watchdogs.

Is-job-running watchdog

The most obvious watchdog is one which checks that there is a training SLURM job running or more are scheduled to run.

Here is an example slurm-status.py that was used during BLOOM-176B training. This watchdog was sending an email if a job was detected to be neither running nor scheduled and it was also piping its check results into the main training's log file. As we used Crontab Emulation, we simply needed to drop slurm-status.slurm into the `cron/cron.hourly/` folder and the previously launched SLURM crontab emulating scheduler would launch this check approximately once an hour.

The key part of the SLURM job is:

```
tools/slurm-status.py --job-name $WATCH_SLURM_NAME 2>&1 | tee -a $MAIN_LOG_FILE
```

which tells the script which job name to watch for, and you can also see that it logs into a log file.

For example, if you launched the script with:

```
tools/slurm-status.py --job-name my-training-10b
```

and the current status report shows:

```
$ squeue -u `whoami` -o "%.10i %9P %20j %.8T %.10M %.8l %.6D %.20S %R"
  JOBID    PARTITION NAME             STATE       TIME   TIME_LIM    NODES  START_TIME
NODELIST(REASON)
    87     prod      my-training-10b  RUNNING 2-15:52:19 1-16:00:00  64    2023-10-07T01:26:28
node-[1-64]
```

then all is good. But if `my-training-10b` job doesn't show the alert will be sent.

You can now adapt these scripts to your needs with minimal changes of editing the path and email addresses. And if it wasn't you who launched the job then replace `whoami` with the name of the user who launched it. `whoami` only works if it was you who launched it.

Is-job-hanging watchdog

If the application is doing `torch.distributed` or alike and a hanging occurs during one of the collectives, it'll eventually timeout and throw an exception, which would restart the training and one could send an alert that the job got restarted.

However, if the hanging happens during another syscall which may have no timeout, e.g. reading from the disk, the application could easily hang there for hours and nobody will be the wiser.

Most applications do periodic logging, e.g., most training log the stats of the last N steps every few minutes. Then one could check if the log file has been updated during the expected time-frame - and if it didn't - send an alert. You could write your own, or use io-watchdog for that (no commit since 2019 - treat as functional-but-unmaintained rather than actively developed).

Low disk space alerts

The next biggest issue is running out of disk space. If your checkpoints are large and are saved frequently and aren't offloaded elsewhere it's easy to quickly run out of disk space. Moreover, typically multiple team members share the same cluster and it could be that your colleagues could quickly consume a lot of disk space. Ideally, you'd have a storage partition that is dedicated to your training only, but often this is difficult to accomplish. In either case you need to know when disk space is low and space making action is to be performed.

Now what should be the threshold at which the alerts are triggered. They need to be made not too soon as users will start ignoring these alerts if you start sending those at say, 50% usage. But also the percentage isn't always applicable, because if you have a huge disk space shared with others, 5% of that disk space could translate to many TBs of free disk space. But on a small partition even 25% might be just a few TBs. Therefore really you should know how often you write your checkpoints and how many TBs of disk space you need daily and how much disk space is available.

Use case: During BLOOM training we wrote a 2.3TB checkpoint every 3 hours, therefore we were consuming 18.4TB a day!

Moreover, often there will be multiple partitions - faster IO partitions dedicated to checkpoint writing, and slower partitions dedicated to code and libraries, and possibly various other partitions that could be in use and all of those need to be monitored if their availability is required for the training not crashing.

Here is another caveat - when it comes to distributed file systems not all filesystems can reliably give you a 100% of disk space you acquired. In fact with some of those types you can only reliably use at most ~80% of the allocated storage space. The problem is that these systems use physical discs that they re-balance at the scheduled periods or triggered events, and thus any of these individual discs can reach 100% of their capacity and lead to a failed write, which would crash a training process, even though `df` would report only 80% space usage on the partition. We didn't have this problem while training BLOOM-176B, but we had it when we trained IDEFICS-80B - 80% there was the new 100%. How do you know if you have this issue - well, usually you discover it while you prepare for the training.

And this is not all. There is another issue of inodes availability and some storage partitions don't have very large inode quotas. Python packages are notorious for having hundreds to thousands of small files, which combined take very little total space, but which add up to tens of thousands of files in one's virtual environment and suddenly while one has TBs of free disk space available, but runs out of free inodes and discovering their training crashing.

Finally, many distributed partitions don't show you the disk usage stats in real time and could take up to 30min to update.

footnote: Use `df -ih` to see the inodes quota and the current usage.

footnote: Some filesystems use internal compression and so the reported disk usage can be less than reality if copied elsewhere, which can be confusing.

So here is fs-watchdog.py that was used during BLOOM-176B training. This watchdog was sending an email if any of the

storage requirements thresholds hasn't been met and here is the corresponding fs-watchdog.slurm that was driving it.

If you study the watchdog code you can see that for each partition we were monitoring both the disk usage and inodes. We used special quota tools provided by the HPC to get instant stats for some partitions, but these tools didn't work for all partitions and there we had to fallback to using `df` and even a much slower `du`. As such it should be easy to adapt to your usecase.

Dealing with slow memory leaks

Some programs develop tiny memory leaks which can be very difficult to debug. Do not confuse those with the usage of MMAP where the program uses the CPU memory to read quickly read data from and where the memory usage could appear to grow over time, but this is not real as this memory gets freed when needed. You can read A Deep Investigation into MMAP Not Leaking Memory to understand why.

Of course, ideally one would analyze their software and fix the leak, but at times the leak could be coming from a 3rd party package or can be very difficult to diagnose and there isn't often the time to do that.

When it comes to GPU memory, there is the possible issue of memory fragmentation, where over time more and more tiny unused memory segments add up and make the GPU appear to have a good amount of free memory, but when the program tries to allocate a large tensor from this memory it fails with the OOM error like:

```
RuntimeError: CUDA out of memory. Tried to allocate 304.00 MiB (GPU 0; 8.00 GiB total capacity;
142.76 MiB already allocated; 6.32 GiB free; 158.00 MiB reserved in total by PyTorch)
```

In this example if there are 6.32GiB free, how comes that 304MiB couldn't be allocated.

One of the approaches my team developed during IDEFICS-80B training where there was some tiny CPU memory leak that would often take multiple days to lead to running out of CPU memory was to install a watchdog inside the training loop that would check the memory usage and if a threshold was reached it'd voluntarily exit the training loop. The next training job would then resume with all the CPU memory reclaimed.

footnote: The reality of Machine Learning trainings is that not all problems can be fixed with limited resources and often times a solid workaround provides for a quicker finish line, as compared to "stopping the presses" and potentially delaying the training for weeks, while trying to figure out where the problem is. For example we trained BLOOM-176B with `CUDA_LAUNCH_BLOCKING=1` because the training would hang without it and after multiple failed attempts to diagnose that we couldn't afford any more waiting and had to proceed as is. Luckily this environment variable that normally is used for debug purposes and which in theory should make some CUDA operations slower didn't actually make any difference to our throughput. But we have never figured out what the problem was and today it doesn't matter at all that we haven't, as we moved on with other projects which aren't impacted by that issue.

The idea is similar to the kill and save switches discussed earlier, but here instead of polling for a specific file appearance we simply watch how much resident memory is used. For example here is how you'd auto-exit if the OS shows only 5% of the virtual cpu memory remain:

```
import psutil
for batch in iterator:
    total_used_percent = psutil.virtual_memory().percent
    if total_used_percent > 0.95:
        print(f"Exiting early since the cpu memory is almost full: ({total_used_percent}%)")
        save_checkpoint()
        sys.exit()

    train_step(batch)
```

Similar heuristics could be used for setting a threshold for GPU memory usage, except one needs to be aware of cuda tensor caching and python garbage collection scheduling, so to get the actual memory usage you'd need to do first run the garbage collector then empty the cuda cache and only then you will get real memory usage stats and then gracefully exit the training if the GPU is too close to being full.

```
import gc
import torch

for batch in iterator:
    gc.collect()
    torch.cuda.empty_cache()

    # get mem usage in GiBs and exit if less than 2GiB of free GPU memory remain
    free, total = map(lambda x: x/2**30, torch.cuda.mem_get_info());
    if free < 2:
        print(f"Exiting early since the GPU memory is almost full: ({free}GB remain)")
        save_checkpoint()
        sys.exit()

    train_step(batch)
```

footnote: don't do this unless you really have to, since caching makes things faster. Ideally figure out the fragmentation issue instead. For example, look up `max_split_size_mb` in the doc for `PYTORCH_ALLOC_CONF` as it controls how memory is allocated. Some frameworks like DeepSpeed solve this by pre-allocating tensors at start time and then reuse them again and again preventing the issue of fragmentation altogether.

footnote: this simplified example would work for a single node. For multiple nodes you'd need to gather the stats from all participating nodes and find the one that has the least amount of memory left and act upon that.

Dealing with forced job preemption

Earlier you have seen how the training can be gracefully stopped with a kill switch solution and it's useful when you need to stop or pause the training on demand.

On HPC clusters SLURM jobs have a maximum runtime. A typical one is 20 hours. This is because on HPCs resources are shared between multiple users/groups and so each is given a time slice to do compute and then the job is forcefully stopped, so that other jobs could use the shared resources.

footnote: this also means that you can't plan how long the training will take unless your jobs run with the highest priority on the cluster. If your priority is not the highest it's not uncommon to have to wait for hours and sometimes days before your job resumes.

One could, of course, let the job killed and hope that not many cycles were spent since the last checkpoint was saved and then let the job resume from this checkpoint, but that's quite wasteful and best avoided.

The efficient solution is to gracefully exit before the hard tile limit is hit and the job is killed by SLURM.

First, you need to figure out how much time your program needs to gracefully finish. This typically requires 2 durations:

1. how long does it take for a single iteration to finish if you have just started a new iteration
2. how long does it take to save the checkpoint

If, for example, the iteration takes 2 minutes at most and the checkpoint saving another 2 minutes, then you need at least 4 minutes of that grace time. To be safe I'd at least double it. There is no harm at exiting a bit earlier, as no resources are

wasted.

So, for example, let's say your HPC allows 100 hour jobs, and then your slurm script will say:

```
#SBATCH --time=100:00:00
```

Approach A. Tell the program at launch time when it should start the exiting process:

```
srun ... torchrun ... --exit-duration-in-mins 5990
```

100h is 6000 minutes and so here we give the program 10 mins to gracefully exit.

And when you start the program you create a timer and then before every new iteration starts you check if the time limit is reached. If it is you save the checkpoint and exit.

case study: you can see how this was set in the BLOOM training job and then acted upon here:

```
        # Exiting based on duration
        if args.exit_duration_in_mins:
            train_time = (time.time() - _TRAIN_START_TIME) / 60.0
            done_cuda = torch.cuda.IntTensor(
                [train_time > args.exit_duration_in_mins])
            torch.distributed.all_reduce(
                done_cuda, op=torch.distributed.ReduceOp.MAX)
            done = done_cuda.item()
            if done:
                if not saved_checkpoint:
                    save_checkpoint_and_time(iteration, model, optimizer,
                                             lr_scheduler)
                print_datetime('exiting program after {} minutes'.format(train_time))
                sys.exit()
```

As you can see since the training is distributed we have to synchronize the exiting event across all ranks

You could also automate the derivation, by retrieving the `EndTime` for the running job:

```
$ scontrol show -d job $SLURM_JOB_ID | grep Time
   RunTime=00:00:42 TimeLimit=00:11:00 TimeMin=N/A
   SubmitTime=2023-10-26T15:18:01 EligibleTime=2023-10-26T15:18:01
   AccrueTime=2023-10-26T15:18:01
   StartTime=2023-10-26T15:18:01 EndTime=2023-10-26T15:18:43 Deadline=N/A
```

and then comparing with the current time in the program and instead setting the graceful exit period. There are other timestamps and durations that can be retrieved as it can be seen from the output.

Approach B.1. Sending a custom signal X minutes before the end

In your sbatch script you could set:

```
#SBATCH --signal=USR1@600
```

and then SLURM will send a `SIGUSR1` signal to your program 10min before job's end time.

footnote: normally SLURM schedulers send a `SIGCONT+SIGTERM` signal about 30-60 seconds before the job's time is up, and just as the time is up it will send a `SIGCONT+SIGTERM+SIGKILL` signal if the job is still running. `SIGTERM` can be caught and acted upon but 30 seconds is not enough time to gracefully exit a large model training program.

Let's demonstrate how the signal sending and trapping works. In terminal `A`, run:

```
python -c "
import time, os, signal

def sighandler(signum, frame):
    print('Signal handler called with signal', signum)
    exit(0)

signal.signal(signal.SIGUSR1, sighandler)
print(os.getpid())
time.sleep(1000)
"
```

it will print the pid of the process, e.g., `4034989` and will go to sleep (emulating real work). In terminal `B` now send `SIGUSR1` signal to the python program in terminal `A` with:

```
kill -s USR1 4034989
```

The program will trap this signal, call the `sighandler` which will now print that it was called and exit.

```
Signal handler called with signal 10
```

`10` is the numerical value of `SIGUSR1`.

So here is the same thing with the SLURM setup:

```
$ cat sigusr1.slurm
#SBATCH --job-name=sigusr1
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --time=0:03:00
#SBATCH --partition=mypartition
#SBATCH --output=%x-%j.out
#SBATCH --signal=USR1@170

srun python -c "
import time, os, signal
```

```
def sighandler(signum, frame):
    print('Signal handler called with signal', signum)
    exit(0)

signal.signal(signal.SIGUSR1, sighandler)
print(os.getpid())
time.sleep(1000)
"
```

In the SLURM script we told SLURM to send the program a signal 170 seconds before its end and the job itself was set to run for 180 secs (3 mins).

When this job has been scheduled:

```
sbatch sigusr1.slurm
```

10 seconds (`180-170`) after the job started, it will exit with the log:

```
58307
Signal handler called with signal 10
```

which means the job had a pid `58307` and it caught `SIGUSR1` (`10`) and it exited.

Now that you understand how this machinery works, instead of immediate `exit(0)` you can set exit-asap flag, finish the currently run iteration, check that the flag is up, save the checkpoint and exit. This is very similar to the code shown in Approach A.

Approach B.2. Choosing which process to send the signal to

Now what if your main program isn't the one launched with `srun` - if you were to use an intermediate launcher like `torchrun` or `accelerate` the above recipe won't work, because most likely `SIGUSR1` won't be propagated from the launcher to its children. In this case we need a slightly more complicated slurm script than the one shown earlier.

We have to replace:

```
#SBATCH --signal=USR1@600
```

with:

```
#SBATCH --signal=B:USR1@600
```

The added `B:` tells SLURM not to send the signal to the `srun` process (launcher) but to the `sbatch` shell.

And now we have to change the end of the SLURM script from a typical launcher-based code like:

```
CMD="python -u -m torch.distributed.run ... train.py ..." # real command here
LOG_FILE=/path/to/logs/main_log.txt
srun --jobid $SLURM_JOBID bash -c "$CMD" 2>&1 | tee -a $LOG_FILE
```

to this:

```
trap 'echo "SIGUSR1 received!"; \
pid=$(pgrep -f "^python.*(accelerate|deepspeed|torchrun|distributed.run)"); \
pgrep -P $pid | xargs -r kill -USR1; \
wait;' SIGUSR1

CMD="python -u -m torch.distributed.run ... train.py ..." # real command here
LOG_FILE=/path/to/logs/main_log.txt
srun --jobid $SLURM_JOBID bash -c "$CMD" 2>&1 | tee -a $LOG_FILE &

wait
```

Since `--signal=B:USR1@600` earlier will now send the signal to the `sbatch` shell we can trap it and do something about it, and that's what the `trap` line does.

The magical code inside the signal handler passed to `trap` finds all processes that are immediate children of any of the launchers like `accelerate`, `deepspeed`, `torchrun` or `torch.distributed.run` and sends the `SIGUSR1` signal to them.

Finally the last change is that in order for `trap` to work we need to run `srun` in the background - so we added `&` at the end of the `srun` command and we needed to add `wait` so that the `sbatch` shell won't exit until `srun` finishes.

Your python code that catches the signal handler remains the same as in Approach B.1.

Here are the important parts of the SLURM script together:

```
$ cat launch.slurm
#!/bin/bash
[...]
#SBATCH --partition=dev
#SBATCH --signal=B:USR1 # Custom preemption signal
[...]

trap 'echo "SIGUSR1 received!"; \
pid=$(pgrep -f "^python.*(accelerate|torchrun|deepspeed|distributed.run)"); \
pgrep -P $pid | xargs -r kill -USR1; wait;' SIGUSR1

CMD="python -u -m torch.distributed.run ... train.py ..." # real command here
LOG_FILE=/path/to/logs/main_log.txt
srun --jobid $SLURM_JOBID bash -c "$CMD" 2>&1 | tee -a $LOG_FILE &

wait
```

And your training loop that may have originally looked like this:

```
$ cat train.py

for batch in dl:
    train_iteration(batch)
```

Now it'll become:

```
$ cat train.py

import signal
import sys

pre_emption_activated = False
def activate_pre_emption(sig, frame):
    global pre_emption_activated
    print("SIGUSR1 received, saving checkpoint")
    pre_emption_activated = True

signal.signal(signal.SIGUSR1, activate_pre_emption)

for batch in dl:
    train_iteration(batch)

    if pre_emption_activated:
        save_checkpoint()
        sys.exit()
```

Of course, you will probably set a flag in the trainer object in the real software and not use a `global`, but for the sake of the short demo that's good enough.

If you want to test this solution, simply change your SLURM script header to:

```
#SBATCH --time=0:05:00
#SBATCH --signal=B:USR1@60
```

Here we tell SLURM to run the job for 5 minutes only (`--time=0:05:00`) and we ask it to send `SIGUSR1` to our `sbatch` script `60` seconds before 5 minutes expires, i.e. 4 minutes after the job started.

QoS-based SLURM preemption

We haven't discussed so far what happens when Quality of Service (QoS) is used, which may also forcefully preempt an existing job. The functionality is the same as job's-allocated-time-is-about-to-end sort of pre-emption, except it can happen any time and not X seconds before the end of the job.

Consider a SLURM setup where you have `--qos=high` which can preempt `--qos=low` jobs and the low priority job has grace time of 10 minutes to shut down:

```
$ sacctmgr show qos format=name,priority,preempt,MaxTRESPerUser,GraceTime,Preempt,Flags
```

```
       Name   Priority     MaxTRESPU  GraceTime    Preempt                Flags
---------- ---------- ------------- ---------- ---------- --------------------
        low          0                 00:10:00
       high          0                 00:00:00        low
```

This is very similar to the time-based pre-emption except here the grace time is hardcoded and can't be modified by the user.

If a job is launched with `--qos=high` and there aren't enough nodes, SLURM will kick out a few low priority jobs to make the nodes available for the high priority job.

By default `GraceTime` could be very short an insufficient for your program to wind down safely if it gets pre-empted - in which case ask your sysadmin to raise its duration to what will work for your needs.

Otherwise the same solutions described in Approaches B.1 and B.2 will work for this type of forced pre-emption.

Reproducibility

Achieve determinism in randomness based software

When debugging always set a fixed seed for all the used Random Number Generators (RNG) so that you get the same data / code path on each re-run.

On GPU, seeding alone isn't enough for bitwise-reproducible results: cuDNN and cuBLAS are free to pick different (but individually valid) algorithms from run to run for the same op, so a truly deterministic re-run also requires forcing them to always pick the same, deterministic ones - that's what the `cudnn.benchmark`, `use_deterministic_algorithms()` and `CUBLAS_WORKSPACE_CONFIG` settings below take care of.

There are many different systems involved, so it's tricky to cover them all - here is an attempt to cover the most common deterministic knobs:

```
import os, random, torch, numpy as np
def enforce_reproducibility(use_seed=None):
    seed = use_seed if use_seed is not None else random.randint(1, 1000000)
    print(f"Using seed: {seed}")

    if use_seed is not None: # slower speed! https://docs.pytorch.org/docs/stable/notes/
randomness.html#cuda-convolution-benchmarking
        # CUBLAS_WORKSPACE_CONFIG must be set before any CUDA context gets created, i.e. before
        # the torch.cuda.* calls below - if CUDA is initialized earlier elsewhere it's too late,
        # and silently so, hence the explicit assert instead of letting it fail quietly
        if "CUBLAS_WORKSPACE_CONFIG" not in os.environ and torch.cuda.is_initialized():
            raise RuntimeError(
                "CUDA was already initialized before enforce_reproducibility() got to set "
                "CUBLAS_WORKSPACE_CONFIG - move this call earlier, or export "
                "CUBLAS_WORKSPACE_CONFIG=:4096:8 before the process starts"
            )
        os.environ.setdefault("CUBLAS_WORKSPACE_CONFIG", ":4096:8")
        torch.use_deterministic_algorithms(True)
        torch.backends.cudnn.benchmark = False

    random.seed(seed)    # python RNG
    np.random.seed(seed) # numpy RNG

    # pytorch RNGs
    torch.manual_seed(seed)          # cpu + cuda
    torch.cuda.manual_seed_all(seed) # multi-gpu - can be called without gpus

    return seed
```

Notes:

- Comment out whichever RNG seeding call you don't need - e.g. if you don't use `numpy` directly. However, some module you depend on might use it internally without your knowledge, so it's usually best to leave them all in and be safe.

- `CUBLAS_WORKSPACE_CONFIG` plugs a gap `use_deterministic_algorithms(True)` doesn't close on its own: cuBLAS - the library behind `torch.mm/mv/bmm` and other `matmul`-based ops - can be non-deterministic on CUDA >= 10.2, because it dynamically picks internal workspace memory for routines running in parallel CUDA streams on the same handle, so results can differ from run to run even with the same seed. Without this env var set, `use_deterministic_algorithms(True)` will raise a `RuntimeError` the moment it hits a `matmul` op - that's your signal it needs to be set. It **must be set before the CUDA context is created**, i.e. before the first CUDA call; if set after CUDA has already been initialized elsewhere in the process, it's silently ignored, which is why it's set at the very top of `enforce_reproducibility()` above, ahead of the `torch.cuda.*` calls.

  Since a silently-ignored env var is easy to miss, `torch.cuda.is_initialized()` is used above to `raise` instead, if `CUBLAS_WORKSPACE_CONFIG` isn't set yet and CUDA has already been lazily initialized by some earlier call (e.g. a `.cuda()` tensor, `torch.cuda.set_device()`, or another library touching CUDA first) by the time `enforce_reproducibility()` runs. Note that this only catches initialization that went through PyTorch's own lazy-init path - if some non-PyTorch library created a raw CUDA context first, `is_initialized()` won't know about it. Therefore, try to run `enforce_reproducibility()` as early as possible in your program.

  NVIDIA's cuBLAS docs sanction exactly two values for deterministic behavior, and both involve a memory-vs-performance tradeoff - a smaller workspace leaves cuBLAS with fewer algorithm choices, which can limit performance, while a larger workspace gives it more room to pick a faster algorithm at the cost of extra GPU memory:

    - `:16:8` - deterministic, minimal extra memory, but the smaller workspace may limit performance
    - `:4096:8` - deterministic, costs ~24MiB of extra GPU memory, but leaves more performance headroom (the more commonly used of the two, and what's used above)

  There's no third, cost-free option - determinism is structurally at odds with letting cuBLAS freely pick whichever algorithm/workspace layout is fastest across concurrent streams, so pinning it down always gives up some of that freedom. `:4096:8` tends to have the smaller performance penalty of the two (more workspace room means fewer algorithms get ruled out), but "smaller" isn't "zero" - the actual hit is workload- and `matmul`-shape-dependent, and can range from negligible to significant (some reports show throughput cut roughly in half on specific workloads).

  The general format is `:[SIZE]:[COUNT]`, and you'll see it chained into multiple `SIZE:COUNT` pairs (e.g. the default, unset behavior is equivalent to `CUBLAS_WORKSPACE_CONFIG=:4096:2:16:8`). That chained form comes from cuBLAS's own internal workspace-pool config, where each pair originally described a distinct bucket size/count so the pool could efficiently serve both large and small workspace requests. PyTorch, however, just sums every `SIZE * COUNT` pair it finds into a single total byte count and hands that one fixed-size buffer to cuBLAS - so `:4096:2:16:8` simply means `2*4096 + 8*16` KiB in total, and chaining more pairs only changes *how many KiB you add up*, not the determinism behavior. In practice you don't need to chain anything yourself: pick one of NVIDIA's two blessed single-pair values above, or set `:0:0` to force cuBLAS to not use any workspace at all (not recommended - this is the most restrictive and slowest option).

- `torch.use_deterministic_algorithms(True)` makes cuDNN convolutions deterministic and it additionally covers hundreds of other ops throughout PyTorch (`index_add_`, `scatter_`, interpolation, etc.). Instead of silently doing its best, it raises a `RuntimeError` the moment it hits an op with no deterministic implementation, so nondeterminism can't sneak in unnoticed.

- `torch.backends.cudnn.benchmark = False` is a separate, complementary knob that stops cuDNN from timing several convolution algorithms on the first call with each new input shape and caching whichever was fastest. That benchmarking step is itself a source of run-to-run variation - which algorithm "wins" the timing race can depend on incidental system noise - independently of whether the algorithm it settles on

is deterministic.

> **PyTorch version note:** `torch.use_deterministic_algorithms()` requires **PyTorch >= 1.8**. On PyTorch < 1.8, fall back to the narrower `torch.backends.cudnn.deterministic = True`, which only makes cuDNN convolutions deterministic and leaves everything else (`matmuls`, `index_add_`, `scatter_`, etc.) untouched - `CUBLAS_WORKSPACE_CONFIG` is independent of the PyTorch version and still applies:
>
> ```
> if use_seed:
>     os.environ.setdefault("CUBLAS_WORKSPACE_CONFIG", ":4096:8")
>     torch.backends.cudnn.benchmark     = False
>     torch.backends.cudnn.deterministic = True # PyTorch < 1.8 fallback - cuDNN
> convolutions only
> ```

Resources:

- NVIDIA cuBLAS docs - Reproducibility - the canonical explanation of cuBLAS non-determinism and the accepted `CUBLAS_WORKSPACE_CONFIG` values
- `torch.use_deterministic_algorithms` - the full list of affected ops and how `warn_only` works
- PyTorch CUDA environment variables reference - the exact `:[SIZE]:[COUNT]` format spec for `CUBLAS_WORKSPACE_CONFIG`
- PyTorch Reproducibility notes - ties this together with seeding, `cudnn.deterministic` and `cudnn.benchmark` for the full picture

If your code also touches other accelerator backends or frameworks besides CPU/CUDA - e.g. NPU, XPU, or TensorFlow - seed their RNGs too:

```
    torch.npu.manual_seed_all(seed)
    torch.xpu.manual_seed_all(seed)
    tf.random.set_seed(seed)
```

When you rerun the same code again and again to solve some problem, set a specific seed at the beginning of your code with:

```
enforce_reproducibility(42)
```

As mentioned above, **this is for debug only**: it activates various `torch` flags that help with determinism, but can slow things down, so you don't want this in production.

However, you can call this instead to use in production:

```
enforce_reproducibility()
```

i.e. w/o the explicit seed. And then it'll pick a random seed and log it! So if something happens in production, you can now reproduce the same RNG state the issue occurred under. And no performance penalty this time, as the determinism settings (`cudnn.benchmark` / `use_deterministic_algorithms` / `CUBLAS_WORKSPACE_CONFIG`) are only set if you provided

the seed explicitly. Say it logged:

```
Using seed: 1234
```

you then just need to change the code to:

```
enforce_reproducibility(1234)
```

and you will get the same RNGs setup.

As mentioned earlier, there could be many other RNGs involved in a system - for example, if you want the data to be fed in the same order for a `DataLoader`, you need to have its seed set as well.

See also Floating point math discrepancies on different devices in the debugging chapter.

Reproduce the software and system environment

This methodology is useful when discovering some discrepancy in outcomes - quality or a throughput for example.

The idea is to log the key components of the environment used to launch a training (or inference) so that if at a later stage it needs to be reproduced exactly as it was it can be done.

Since there is a huge variety of systems and components being used it's impossible to prescribe a way that will always work. So let's discuss one possible recipe and you can then adapt it to your particular environment.

This is added to your slurm launcher script (or whatever other way you use to launch the training) - this is Bash script:

```
SAVE_DIR=/tmp # edit to a real persistent path
export REPRO_DIR=$SAVE_DIR/repro/$SLURM_JOB_ID
mkdir -p $REPRO_DIR
# 1. modules (writes to stderr) (remove if you don't use lmod or similar modules implementation)
module list 2> $REPRO_DIR/modules.txt
# 2. shell env vars
/usr/bin/printenv | sort > $REPRO_DIR/env.txt
# 3. pip (this includes devel installs SHA)
pip freeze > $REPRO_DIR/requirements.txt
# 4. uncommitted diff in git clones installed into conda
perl -nle 'm|"file://(.*?/([^/]+))"| && qx[cd $1; if [ ! -z "\$(git diff)" ]; then git diff >
\$REPRO_DIR/$2.diff; fi]' $CONDA_PREFIX/lib/python*/site-packages/*.dist-info/direct_url.json
```

As you can see this recipe is used in a SLURM environment, so every new training will dump the environment specific to the SLURM job. But you can adapt it to any other environment.

Let's expend on each step in the recipe:

1. We save which `modules` were loaded, e.g. in cloud cluster/HPC setups you're like to be loading the CUDA and cuDNN libraries using this.

   If you don't use `modules` then remove that entry

2. We dump the shell environment variables. This can be crucial since a single env var like `LD_PRELOAD` or

`LD_LIBRARY_PATH` could make a huge impact on performance in some environments

3. We then dump the python environment packages and their versions - this should work with any virtual python environment like `conda`, `venv` or even if you don't use a virtual environment.

   If you use `uv` instead of `pip`, switch to `uv pip freeze` as it'd be much faster.

4. If you use a devel install with `pip install -e .` it doesn't know anything about the git clone repository it was installed from other than its git SHA. But the issue is that it's likely that you have modified the files locally and now `pip freeze` will miss those changes. So this part will go through all packages that are not installed into the conda environment (we find them by looking inside `site-packages/*.dist-info/direct_url.json`) and, for each one that has uncommitted changes, save its `git diff` to `$REPRO_DIR/<package>.diff`.

To save the `apt` packages add:

```
apt list --installed > $REPRO_DIR/apt-packages.txt
```

If using `conda` an additionally useful tool is conda-env-compare.pl which helps you to find out the exact differences 2 conda environments have.

Anecdotally, me and my colleague were getting very different training TFLOPS on a cloud cluster running the exact same code - literally launching the same slurm script from the same shared directory. We first compared our conda environments using conda-env-compare.pl and found some differences - I installed the exact packages she had to match her environment and it was still showing a huge performance difference. We then compared the output of `printenv` and discovered that I had `LD_PRELOAD` set up whereas she didn't - and that made a huge difference since this particular cloud provider required multiple env vars to be set to custom paths to get the most of their hardware.

Avoiding, Recovering From and Understanding Instabilities

Sub-sections:

- Understanding Training Loss Patterns - types of spikes, divergences, grokking moments, resumes, etc.

Learning from Training Logbooks

The best learning is to read Publicly available training LLM/VLM logbooks because there you can see exactly what happened and how the problem has been overcome.

STD Init

Correctly initializing the initial distribution of the tensors can have a tremendous impact on training's stability. The `std` value isn't fixed and depends on the hidden dimension size.

This proved to be a very crucial setting in our pre-BLOOM 104B experiments and we couldn't break past the first few thousands iterations until we figured out that the 0.02 default `--init-method-std` in Megatron-LM was a way too big for our model.

We referred to these two sources:

1. "Transformers without Tears" paper https://arxiv.org/abs/1910.05895 prescribes: `sqrt(2/(NHIDDEN*5))`
2. The 530B training paper https://arxiv.org/abs/2201.11990 they used an even smaller init formula: `sqrt(1/(NHIDDEN*3))`

and decided to go with the 530B one as it leads to an even smaller init value.

To make it easier to compare the two formulas, they can be rewritten as:

1. `sqrt(0.4000/NHIDDEN)`
2. `sqrt(0.3333/NHIDDEN)`

Thus for `NHIDDEN=14336` the math was `sqrt(1/(14336*3)) = 0.00482` and that's what we used. It surely wasn't the only reason why we had no stability issues during BLOOM-176B training, but I think it was one of the crucial ones.

Numerical instabilities

See also Detecting problematic tensor values and Underflow and Overflow Detection in the debugging chapter for tooling to locate `inf`/`nan` values.

Certain mathematical operations could be unstable when dealing with low precision numbers.

For example, please see this very interesting PyTorch guide on numerical stability.

Now let's look at a specific example of this concept in action.

During 104B training experiments where fp16 mixed precision was used - the following improvement was proposed by Corby Rosset to make self-attention more stable.

Specifically this line shows that the `norm_factor` may be multiplied after the Query * Key matrix multiplication. If the dim of Q and K are very large, the output may blow up and the `norm_factor` won't be able to save it.

Proposal: move the `norm_factor` inward, so Q and K are scaled down before matrix multiply:

```
        matmul_result = torch.baddbmm(
            matmul_result,
            1.0/math.sqrt(self.norm_factor) * query_layer.transpose(0, 1),   # [b * np, sq, hn]
            1.0/math.sqrt(self.norm_factor) * key_layer.transpose(0, 1).transpose(1, 2),  # [b *
np, hn, sk]
            beta=0.0 if alibi is None else 1.0, alpha=1.0)

        # change view to [b, np, sq, sk]
        attention_scores = matmul_result.view(*output_size)
```

To make the operation mathematically equivalent, moving the norm factor inward requires taking sqrt again if n is a scalar, A and B matrices:

```
n * (A dot B) === (sqrt(n) * A) dot (sqrt(n) * B)
```

Now A and B dimensions can be significantly larger.

The same post-multiply scaling problem also exists one level down, in the CUDA matmul APIs themselves.

For CUDA kernel writers CuBlas's `GemmStridedBatchedEx` at the time of this writing has a similar issue. It is defined as:

```
C+i*strideC=αop(A+i*strideA)op(B+i*strideB)+β(C+i*strideC), for i ∈[0,batchCount−1]
```

The issue is that `alpha` is multiplied after the matrix-matrix multiplication is done so it can cause instability.

"Bad" combination of data batch and model parameter state

PaLM team observed dozens of loss spikes at "highly irregular intervals" when training larger models. While they were not able to track down the root cause, they mitigated the issue by restarting from an earlier checkpoint and skipping potentially problematic data batches. Section 5.1 Training instability

Time-domain correlation divergence in Adam

A Theory on Adam Instability in Large-Scale Machine Learning performs a rigorous study of divergence spikes while training LLMs at up to 546B parameters - and suggests that the time-domain correlation leads to divergence of Adam. This is triggered by the epsilon value not being small enough and gradient estimation components become similar to the epsilon.

In section 7.1 they propose practical suggestions, the most interesting one of them is setting epsilon to 0 and possibly dealing with division by zero condition.

Understanding Training Loss Patterns

Training loss plot is similar to the heart beat pattern - there is the good, the bad and you-should-worry one. After studying many training loss trajectories one develops an intuition to explain various loss behaviors during one's training and how to act on those.

I warn you that the "Understanding" in the title of this section is overloaded since very often we don't really understand why certain types of spikes happen. Here "understanding" refers to recognizing various patterns. We then usually have techniques to overcome the bad patterns and bring the training successfully to the finish line.

Thus you will find here a gallery of training loss patterns sometimes with real explanations, but more often than not educated guesses to what might be happening.

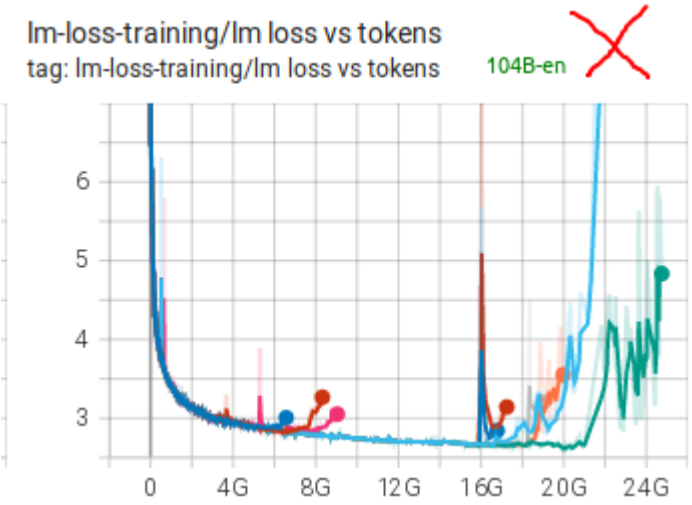
Please excuse the plot snapshots looking wildly different from each other as they have come from many sources over multiple years.

The good, the bad and the unexpected

Let's look at some good, bad and unusual patterns.

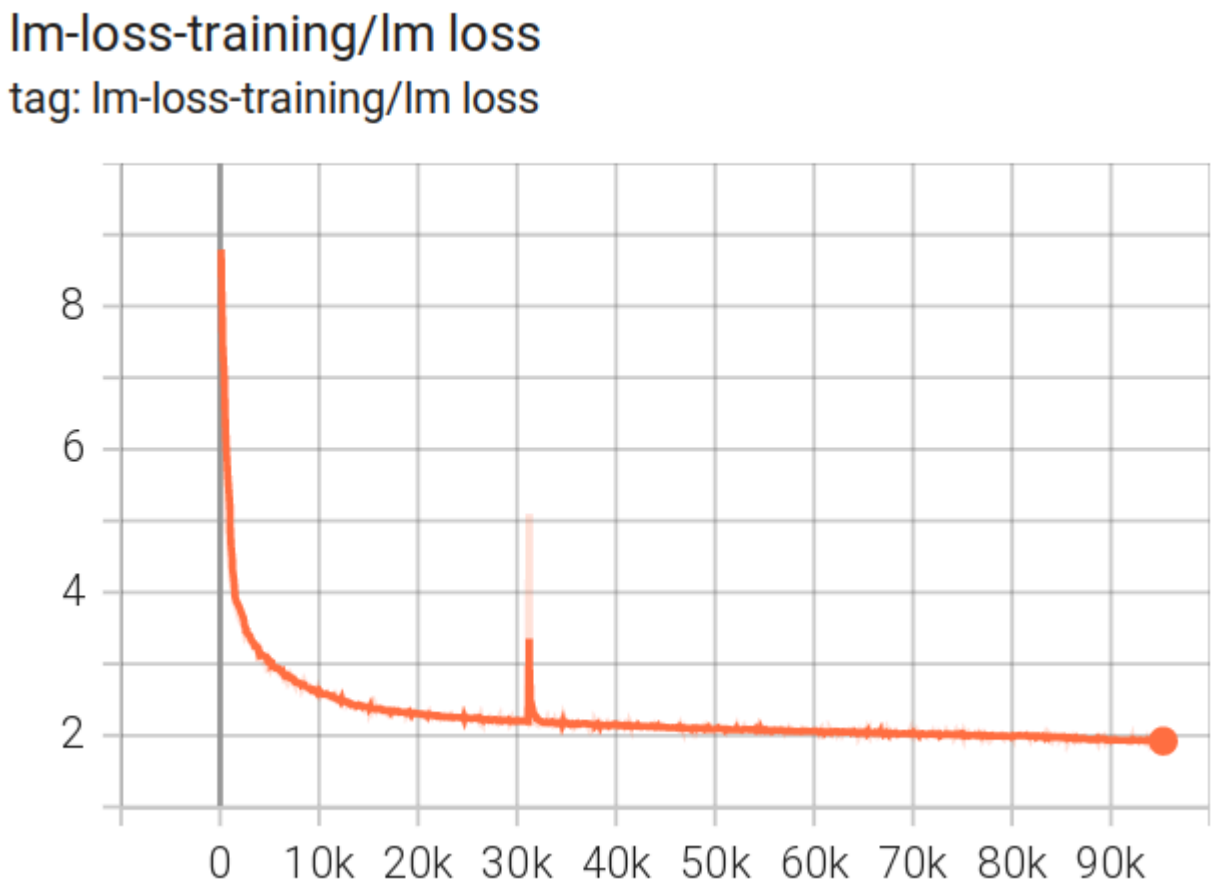
A very failed training

Prior to starting BLOOM-176B training we did multiple experiments with the 104B model. We failed to figure out how to not diverge very early on.



As you can see many attempts were made, many techniques were applied (see chronicles. We think the 2 main obstacles were using fp16 and data that had a lot of garbage in it. For BLOOM-176B we switched to bf16, used much cleaner data and also added an embedding layer-norm and that made all the difference.

An almost perfect training



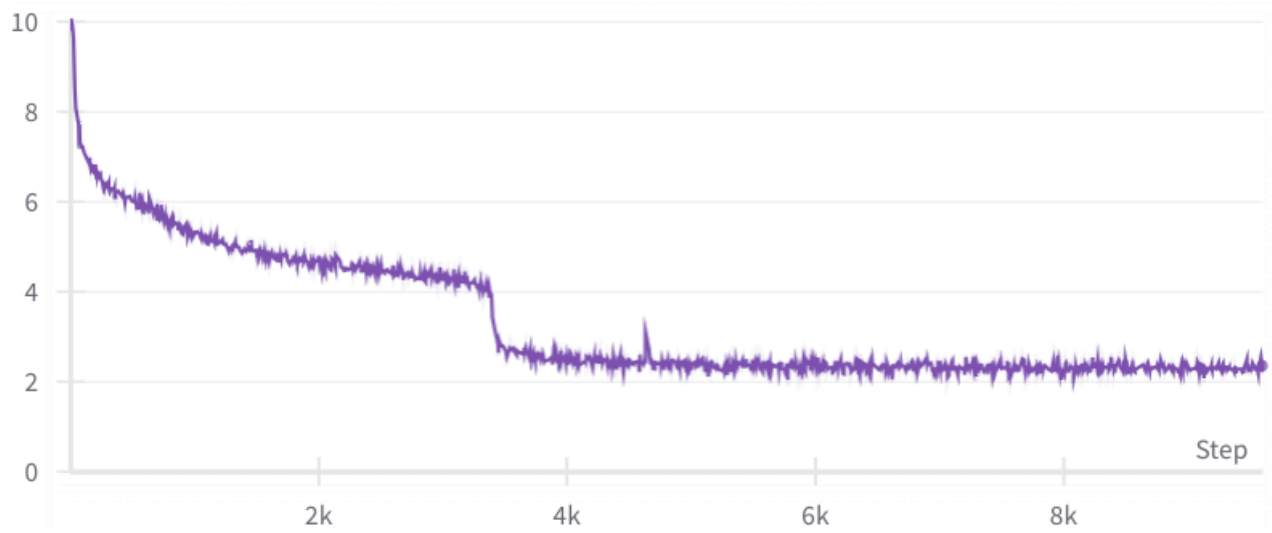
The BLOOM-176B training had a close to perfect training loss trajectory, with a single spike that has recovered in 200 steps.

You can inspect the TB to zoom in and check other plots.

This was the almost perfect training indeed. Lots of hard work was put into achieving this.

The grokking moment

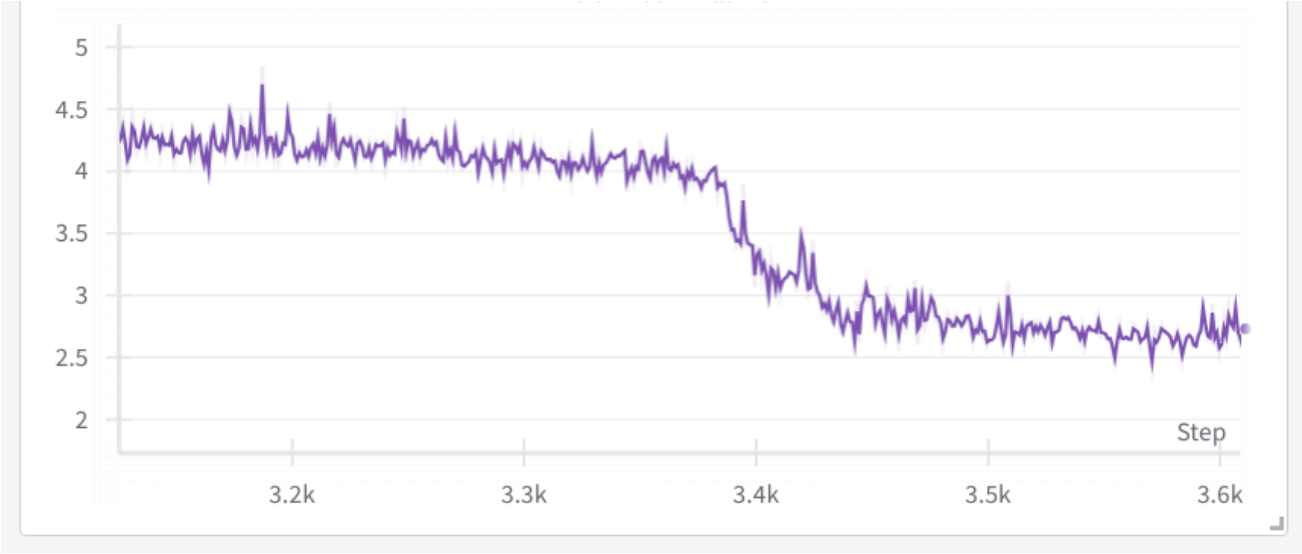
Some time back I was doing performance testing and run a tiny global batch size of 8 on an 8×A100 node, training llama-2-7b from scratch. (w/ DeepSpeed ZeRO-3 DP using HF Transformers Llama implementation)



Here one can observe a rapid loss improvement from 4 to 2.5 in just 480 samples after a very steady much slower improvements. My colleague Gautam Mittal called it the grokking moment. In just a handful of steps the model suddenly generalized to much better predict the next tokens.

Normally one doesn't see such a dramatic improvement when using a much larger batch size.

If we zoom in it took about 60 8-sample per iteration steps:



Main types of loss spikes

In general there are 3 types of loss spikes:

1. Fast recovering spikes
2. Slow recovering spikes
3. Not fully recovering spikes

The spikes usually happen because of a bad data pocket, either due to badly shuffled data or because it hasn't been cleaned from some garbage scraped from the websites.

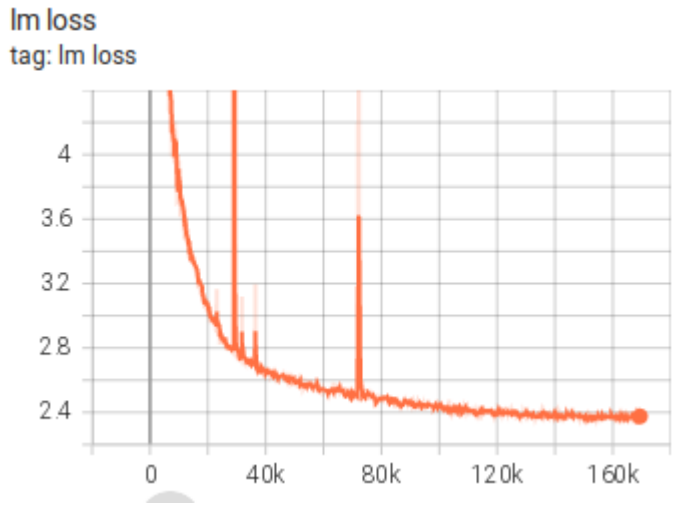
While one would suspect that the batch before the spike was the trigger, but if you were to study that batch's contents

you are likely to find nothing unusual - quite often the problem starts developing many steps before and then most of the sudden it happens. But also it might not be easy to study the batch, since it could amount to a size of a book when the global batch size and the sequence lengths are huge.

Fast recovering spikes

Loss spikes can happen often and as long as they quickly bounce back to where they left off the training usually continues as if nothing happened:

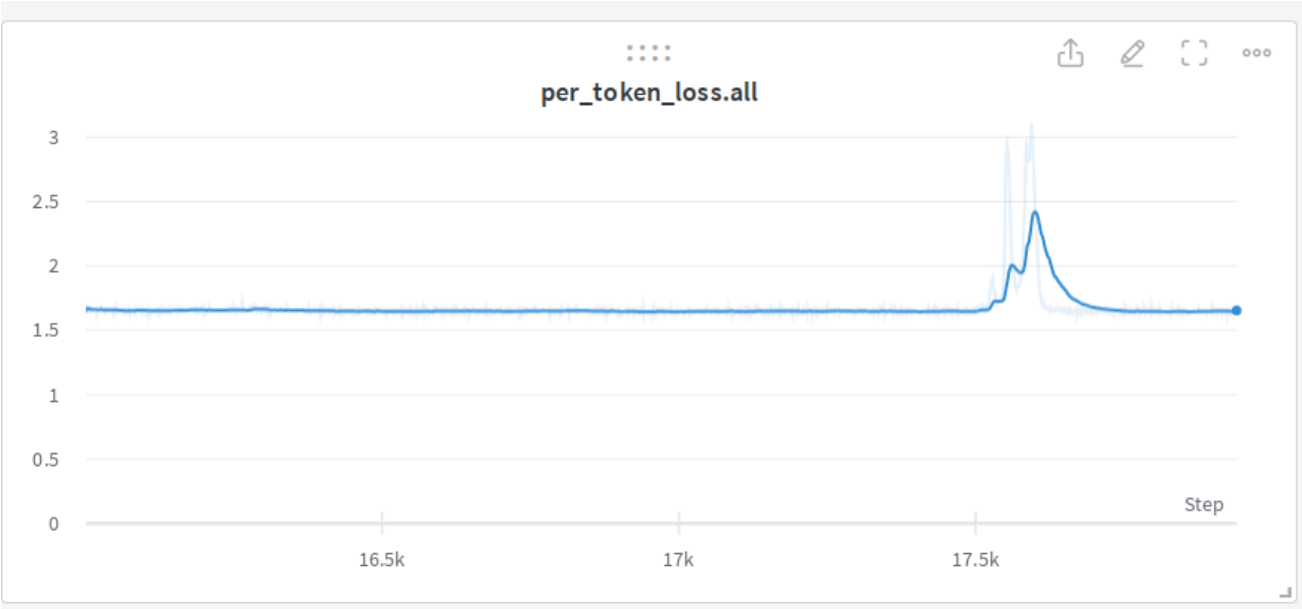
Here is an example of the 13B pre-BLOOM training experiment:



As you can see there are many spikes, some of a huge magnitude but they have all quickly recovered.

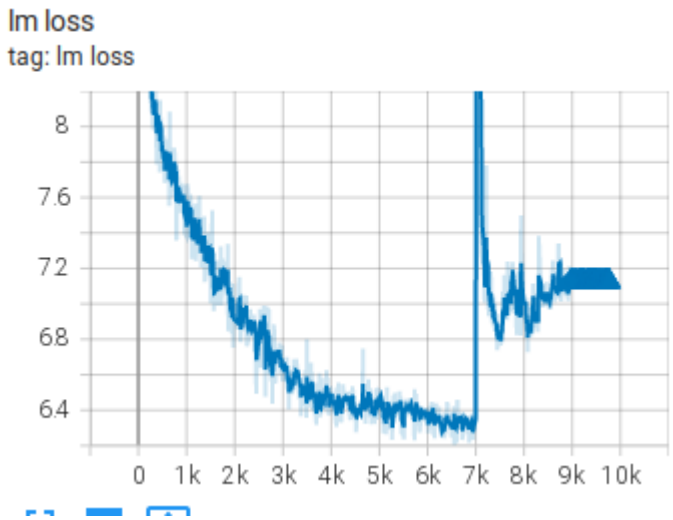
Slow recovering spikes

Here is a slow recovering spike from the IDEFICS-80B training:

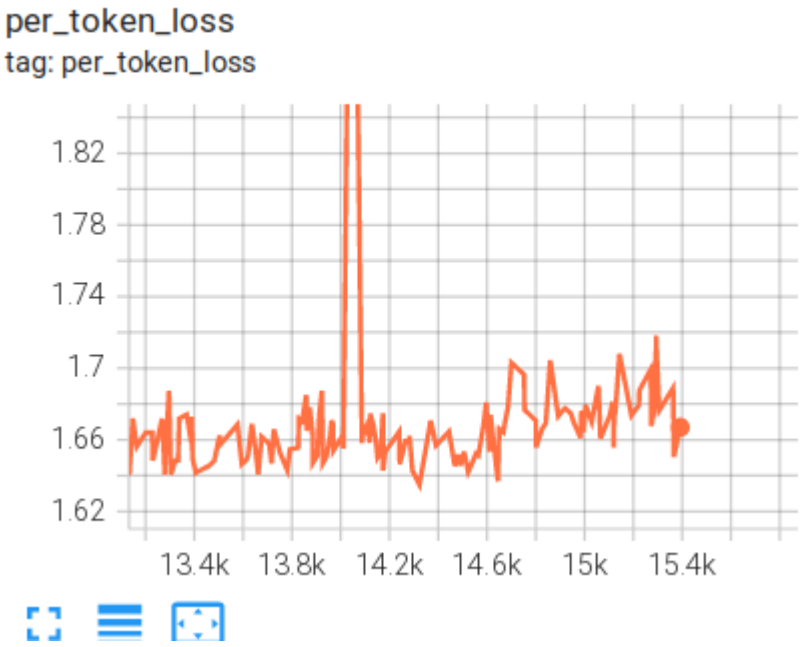


Not fully recovering spikes

This 104B model attempt spiked, started recovering but decided to not recover fully and instead started diverging

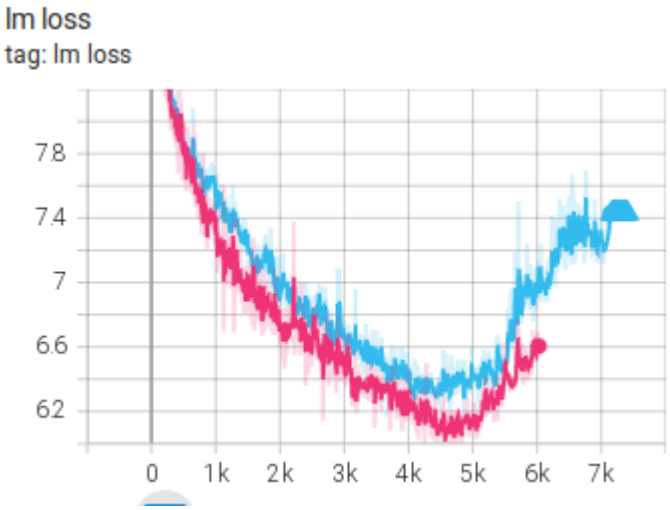


Here is another example from the IDEFICS-80B training:

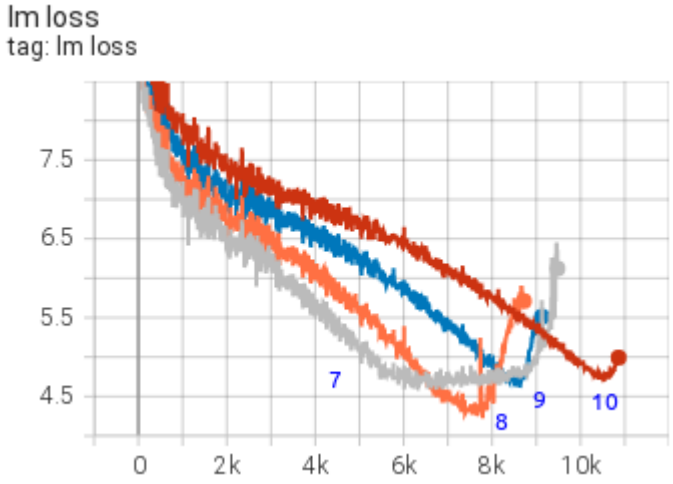


Non-spike diverging

Here are a few examples of diverging that didn't go through a spike



and here are a few more:

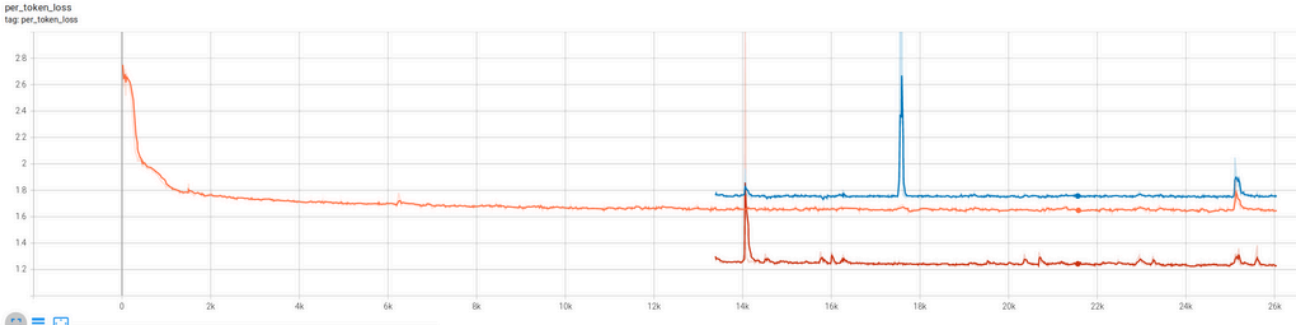


as you can see each restart makes a bit of progress and then the model diverges.

All these are from the 104B model attempts.

Multiple datasets spikes

During the IDEFICS-80B training we were using 2 different dataset types mixed together:



Legend: cm4 (high), average (mid) and pmd (low)

You can see that the loss spikes were sometimes happening simultaneously on both datasets and at other times only one

of the datasets loss would spike.

Here the model was learning two different data distributions and as you can see it was not reporting the same loss and the spike behaviors on both data distributions. The pmd datasets loss was much easier for the model than the cm4 one.

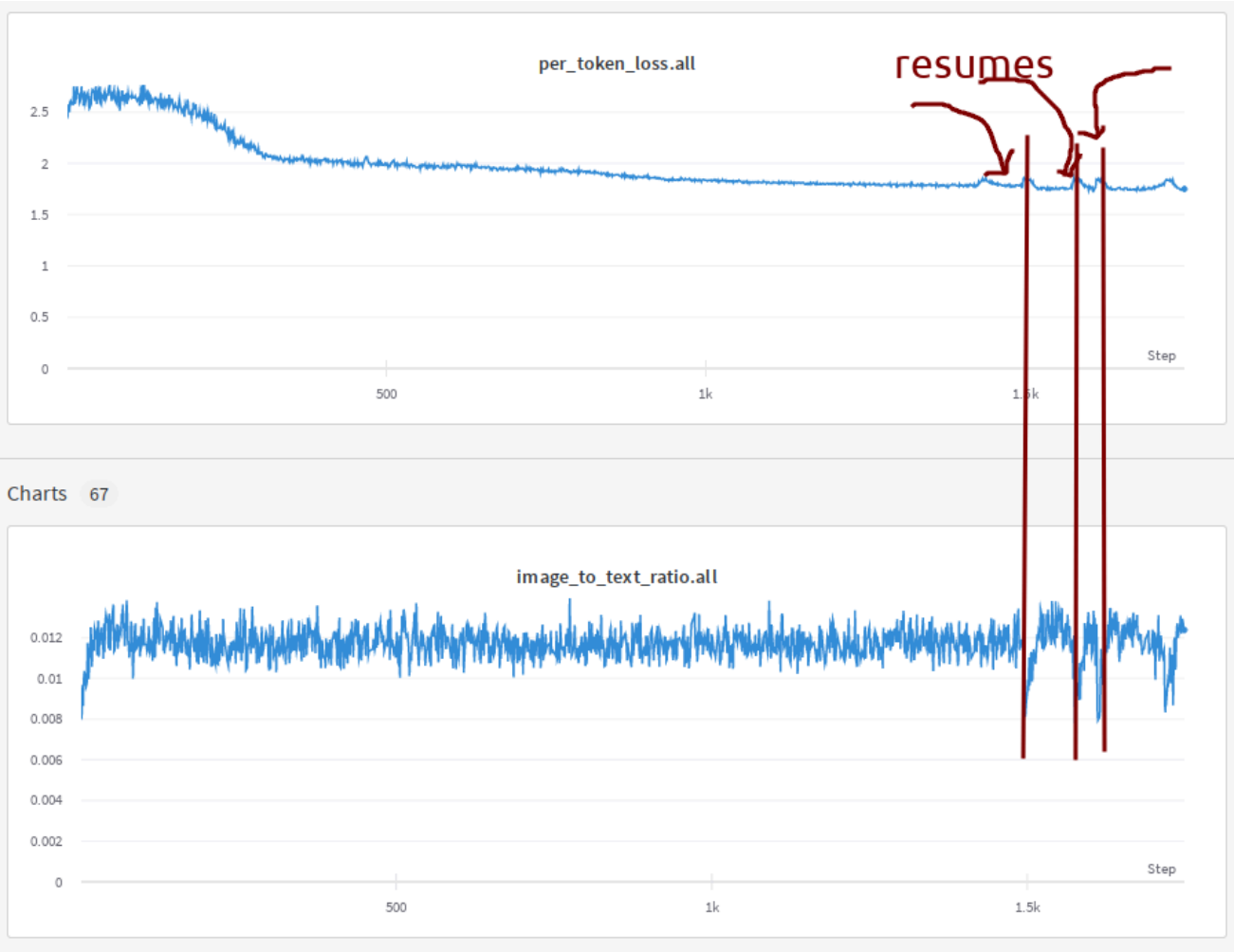
Resume-related spikes

Training resume due to a hardware crash or because a need to rollback to an earlier checkpoint due to encountering a divergence is pretty much guaranteed to happen. If your training software doesn't resume perfectly so that the model doesn't notice there was a resume various problems could be encountered.

The most complicated challenge of resume is restoring various RNGs, getting to the DataLoader index where the previous training was restored, and dealing with various other requirements if you use complex DataLoaders that are specific to your setup.

DataSampler related issues

During IDEFICS-80B training we had a very complicated DataLoader which was suffering from image to text ratio fluctuations when the DataLoader was getting restored on resume, so we ended up having a small spike on each resume which would then recover:



You can see the loss and ratio plots correlation here. As we had to resume about a dozen times we saw a lot of those spikes.

Impacts of repeat data

I was training a variation of Llama2 and saw this super unusual spike that didn't diverge or recover but which switched to a new higher loss level:

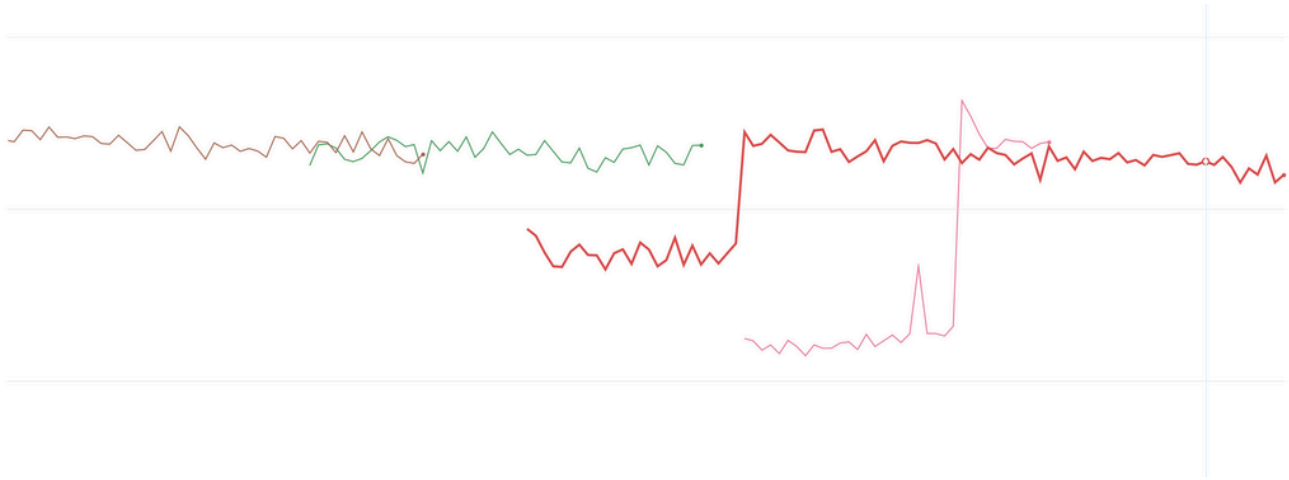


I rolled back to just before the weird behavior occurred and restarted. The loss training progressed at the same loss level for a bit and then again spiked and shifted to a higher loss.



I have never seen this type of divergence before. I was scratching my head for a while and then decided to look at the bigger picture.

wandb didn't handle resume data plotting correctly if a rollback was performed, that is it ignored all new data after the rollback until the steps of the old data have been overcome. This forced us to start a new wandb plot for every resume with a rollback so that new data is shown. And if you need to see the whole plot you have to stitch them together, which includes dead data points that are no longer true. So I did the stitching and saw this puzzle:



footnote: as of 2025 wandb can do this properly - rewinding a run with `wandb.init(resume_from="<run_id>?_step=N")` truncates the history at step `N` and lets you log forward under the same run id, and `fork_from` does the same while leaving the original run intact (wandb SDK 0.17.1+; wandb recommends forking over rewinding for performance). Two catches: it's cloud-only - Multi-tenant and Dedicated Cloud, not Self-Managed - and it needs monotonically increasing steps, so it won't work alongside a non-monotonic `define_metric()`. Self-hosting, the stitching above is still the way.

There was no real spike in the two earlier runs. The loss never went up in the first place. In both resumes it was under-reporting loss due to an exactly repeated data and then it reached data it hasn't seen before and started reporting

correctly. In other words it was overfitting and reporting a false loss.

The cause of the problem is data repetition, and since it clearly memorised some of it, it was reporting a better loss.

The problem came from pytorch-lightning not handling resumes correctly wrt DataSampler automatically - basically every time you resume you start your data stream from scratch. This, of course, requires a user to somehow fix the situation. You could change the seed to somewhat ameliorate the situation and avoid the exact data sequence, but it still leaves you with repeat data, which isn't what you want for any serious training (or ablation experiments, since your observation will be invalid, if they assume IID data distribution).

footnote: I discussed this issue with the PTL developers and they said that they tried hard to come up with a generic solution but it wasn't meant to be. So the user needs to figure it out.

Make sure to check your training framework documentation whether it handles the DataSampler resuming correctly. Make sure you didn't discover this problem after the training has finished and you ended up training 6x times the same 50B of tokens from the planned 300B tokens seen only once each.

Doing a couple of resumes early on before embarking on the real training should also expose if there is a problem. Albeit, if the data gets reshuffled on each resume you are unlikely to see it. It'll only be seen if the seed is the same.

Checkpoints

- torch-checkpoint-convert-to-bf16 - converts an existing fp32 torch checkpoint to bf16. If safetensors are found those are converted as well. Should be easily adaptable to other similar use cases.
- torch-checkpoint-shrink.py - this script fixes checkpoints which for some reason stored tensors with storage larger than their view at the moment of saving. It clones the current view and re-saves them with just the storage of the current view.

Selecting Training Hyper-Parameters And Model Initializations

The easiest way to find a good hparam and model init starter set is to steal it from a similar training that you know has succeeded. Here is a collection of public training LLM/VLM logbooks to get you started. The other common source is papers if they disclose that information. You can also try to reach out to the authors and ask them for these details if they didn't publish it.

Glossary

Training jargon uses a multitude of abbreviations and terms, so here are some important for this chapter.

- BS: Batch Size - here we mean batch size per gpu, often it is also referred to as MBS (micro-batch-size)
- GAS: Gradient Accumulation Steps - how many forward/backward cycles to perform before one full iteration is complete
- GBS: Global Batch Size - total batch size per iteration - may include gradient accumulation
- PP: Pipeline Parallelism
- TFLOPS: Trillion FLOPs per second - FLOPS

Global Batch Size Ramp Up

If you intend to train with a very large GBS, with say 1024, or 2048 samples and even higher, when you just start training, it's very wasteful to feed such large batch sizes to the model. At this point it's totally random and can't benefit from having too refined data. Therefore to save data and resources, one often ramps up the global batch size over some period of time.

It's also important to not start with GBS that is too small, since otherwise the progress won't be efficient. When there is too little data the compute (TFLOPS) is inefficient and will slow everything down. This is especially so when Pipeline Parallelism (PP) is used, since the most important thing about PP tuneup is a small GPU idleness bubble, and the smaller the GBS the larger the bubble is.

For example, for BLOOM-176B, where we did use PP, after doing throughput benchmarking we found that starting with GBS=16 was incredibly slow (8TFLOPS), so we eventually started with GBS=192 (73TFLOPS) and then we ramped up to GBS=2048 (150TFLOPS) - we increased GBS by 16 every 9_765_625 samples.

STD Init

This hyper parameter is super-important and it requires math to get it right. For details see STD Init.

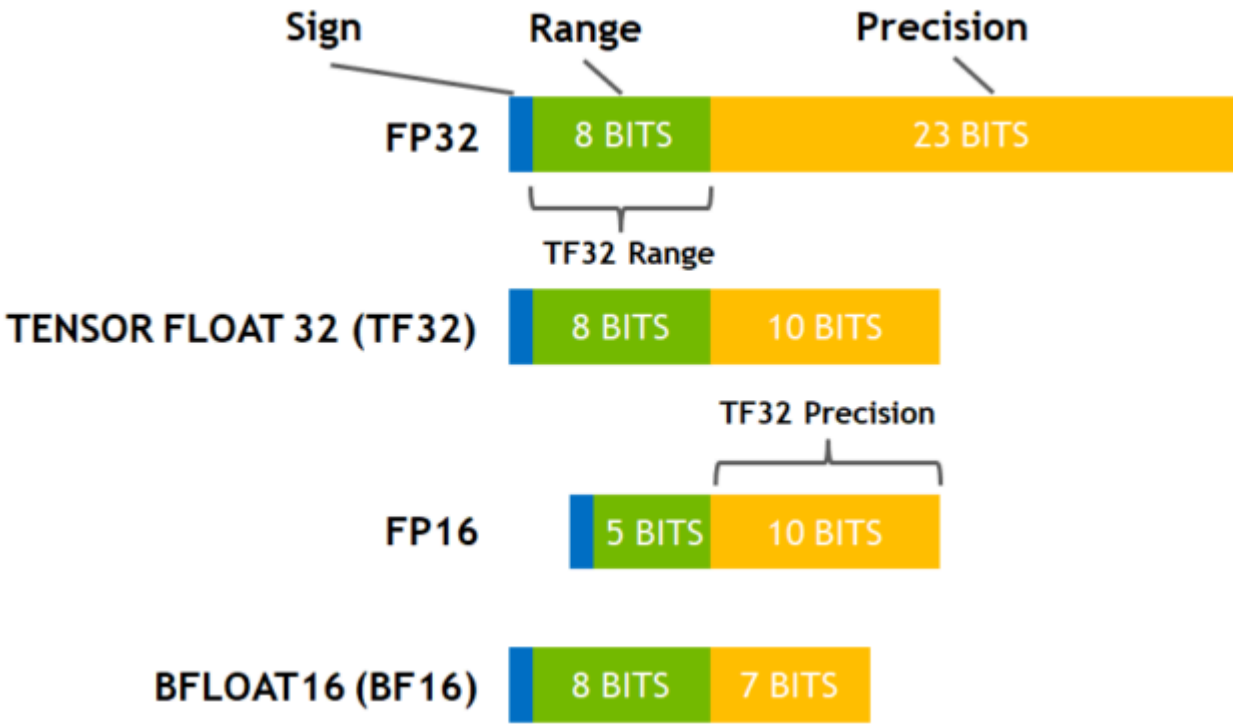
Tensor Precision / Data Types

These are the common data types that are used as of 2026-08 in ML (usually referred to as `dtype`):

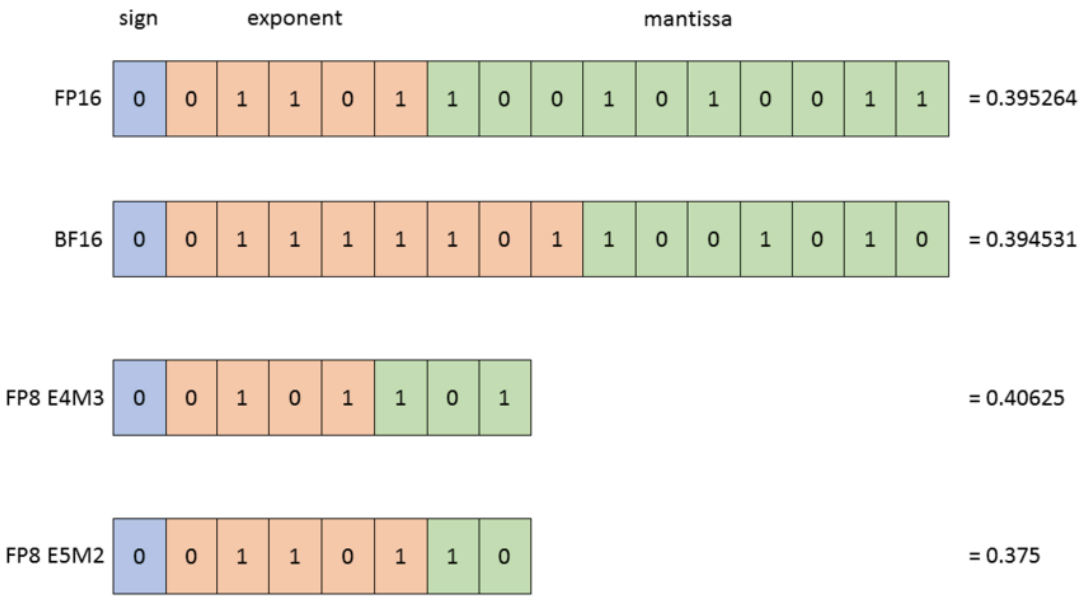
Floating point formats:

- fp32 - 32 bits
- tf32 - 19 bits (NVIDIA Ampere+)
- fp16 - 16 bits
- bf16 - 16 bits
- fp8 - 8 bits (E4M3 and E5M2 formats)
- fp6 - 6 bits
- fp4 - 4 bits

For visual comparison refer to this representations:



(source)



(source)

The new formats that are being adopted by new hardware are:

- fp4: `float4_e2m1fn`
- fp6: `float6_e2m3fn` and `float6_e3m2fn`
- fp8: `float8_e3m4`, `float8_e4m3`, `float8_e4m3b11fnuz`, `float8_e4m3fn`, `float8_e4m3fnuz`, `float8_e5m2`, `float8_e5m2fnuz`, `float8_e8m0fnu`

There is an excellent explanation of each of these variations here.

`float8_e8m0fnu` is the odd one out in that list - it is not an element format but the shared scale that block-scaled formats attach to a group of elements, which is why it has no mantissa. Pairing it with a 4-, 6- or 8-bit element encoding is what produces the `MXFP4`/`MXFP6`/`MXFP8` names quoted in accelerator specs: one `E8M0` scale per block of 32 values, whose per-parameter byte cost is worked out in Model Weights. NVIDIA's `NVFP4` is the same two-part idea tuned differently - the same `E2M1` element, but a block of 16 with an `E4M3` scale plus a single fp32 scale for the whole tensor, which tracks local dynamic range more closely than a power-of-two `E8M0` scale can.

To decipher the letters followed by the numbers:

- The `e` indicates the length of exponent
- The `m` indicates the length of mantissa
- The `b` indicates the bias

To decipher the letters appearing after the numbers:

- The `f` indicates it is finite values only (no infinities).
- The `n` indicates it includes NaNs, but only at the outer range.
- The `u` stands for unsigned format.
- The `uz` stands for unsigned zero.

So for example: `float8_e4m3b11fnuz` stands for fp8 + 4-bit exponent + 3-bit mantissa + bias 11 + finite values only + includes NaNs, but only at the outer range + unsigned zero.

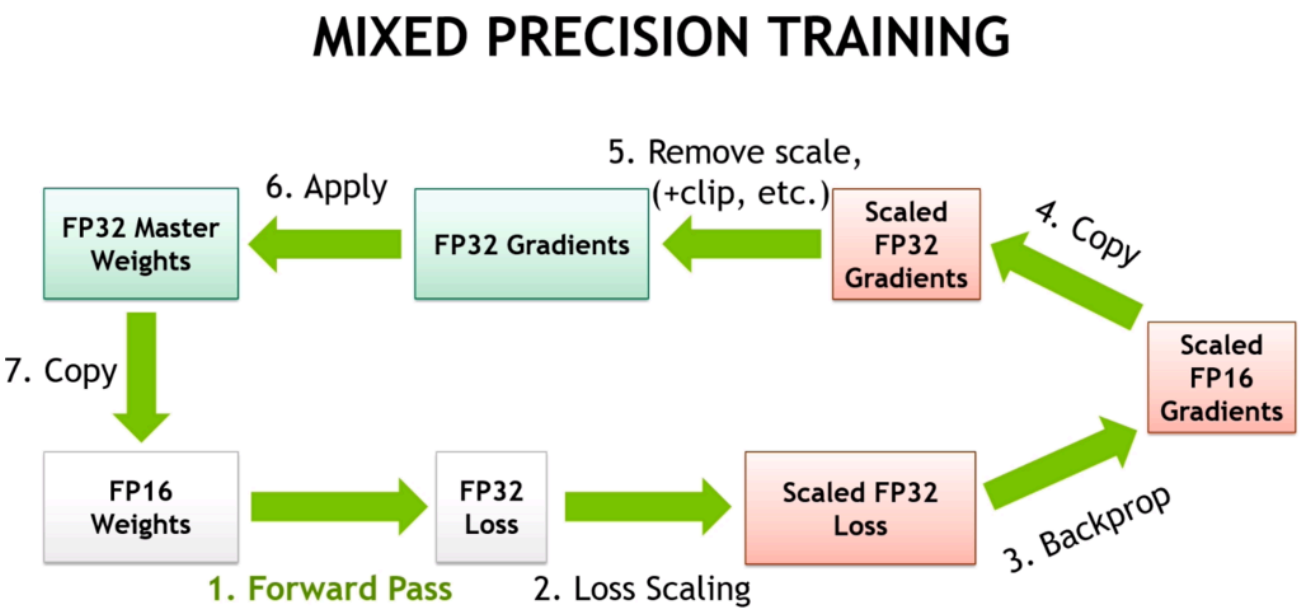
Integer formats used in quantization:

- int8 - 8 bits
- int4 - 4 bits
- int1 - 1 bit

ML dtype progression

Originally ML was using fp32, but it was very slow.

Next mixed-precision was invented using a combination of fp16 and fp32, which tremendously sped up the training speed.



(source)

But fp16 proved to be not very stable and training LLM was extremely difficult.

Luckily bf16 came out and replaced fp16 using the same mixed precision protocol. This made the LLM training much more stable.

Then fp8 came and mixed precision could switch to that, which makes the training even faster. See FP8 Formats for Deep Learning. As of 2026-08 bf16 mixed precision is still the default for most training runs, but fp8 training is no longer experimental - DeepSeek-V3 was trained in fp8.

And then Blackwell added fp6, fp4 and NVIDIA's own nvfp4. So far these are mostly inference formats - fp4 training is still a research topic.

To appreciate the speed ups between the different formats here is a table for NVIDIA B200 TFLOPS spec (w/o sparsity), along with the accelerator that first supported each dtype in non-CPU hardware (cpu is weak for deep learning). It's sorted by `B200 TFLOPS` ascending, and where several dtypes run at the same speed, by the year the hardware support arrived:

Data type	B200 TFLOPS	First hardware support
fp32	80	predates ML accelerators
tf32	1125	NVIDIA A100 (Ampere, 2020)
fp16	2250	NVIDIA P100 (Pascal, 2016)
bf16	2250	Google TPU v2 (2017) NVIDIA A100 (Ampere, 2020)

Data type	B200 TFLOPS	First hardware support
int8	4500	Google TPU v1 (2015) NVIDIA P4/P40 (Pascal, 2016)
fp8	4500	NVIDIA H100 (Hopper, 2022) Intel Gaudi 2 (2022)
fp6	4500	NVIDIA B200 (Blackwell, 2024)
fp4	9000	NVIDIA B200 (Blackwell, 2024)
nvfp4	10000	NVIDIA B200 (Blackwell, 2024)

int8 is TOPS rather than TFLOPS, since those are integer ops.

Some of these dates mark when a dtype became usable rather than when it became fast. P100 could do fp16 arithmetic at 2x the fp32 rate, but it took V100's tensor cores in 2017 to make fp16 `matmul`s fast. Pascal's int8 was the `DP4A` dot-product instruction; int8 tensor cores arrived with Turing in 2018.

The doubling holds all the way down to fp8 - each halving of the element width buys about 2x the throughput. Then it stops: fp6 runs at the same 4500TFLOPS as fp8, so it buys you memory and bandwidth but no compute. fp4 doubles again over fp8, and on GB300 it's 3x (15000 vs 5000). So don't assume the pattern continues - check the spec for the dtype you're actually planning to use.

In parallel with the mixed training regime the ML community started coming up with various quantization approaches. Probably one of the best examples is Tim Dettmers' bitsandbytes which provides many 4 and 8-bit quantization solutions. DeepSpeed also has some interesting quantization solutions.

TF32

TF32 is a magical datatype that is available on NVIDIA GPUs since Ampere, and which allows fp32 `matmul`s performed at a much faster speed than normal fp32 `matmul`s with a small precision loss.

Here is an example of A100 TFLOPS (w/o sparsity):

Data type	TFLOPS
FP32	19.5
Tensor Float 32 (TF32)	156

As you can see TF32 is 8x faster than FP32!

It's disabled by default. To enable it add at the beginning of your program:

```
torch.backends.cuda.matmul.allow_tf32 = True
torch.backends.cudnn.allow_tf32 = True
```

For more information about the actual precision loss please see this.

When to use fp32 accumulators

Whenever a low-precision dtype is used one has to be careful not to accumulate intermediary results in that dtype.

`LayerNorm`-like operations must not do their work in half-precision, or they may lose a lot of data. Therefore when these operations are implemented correctly they do efficient internal work in the dtype of the inputs, but using the fp32 accumulation registers and then their outputs are downcast to the precision of the inputs.

Generally it's just the accumulation that is done in fp32, since adding up many low-precision numbers is very lossy otherwise.

Here are some examples:

1. Reduction collectives

- fp16: ok to do in fp16 if loss scaling is in place
- bf16: only ok in fp32

2. Gradient accumulation

- best done in fp32 for fp16 and bf16, but definitely is a must for bf16

3. Optimizer step / Vanishing gradients

- when adding a tiny gradient to a large number, that addition is often nullified therefore typically fp32 master weights and fp32 optim states are used.
- fp16 master weights and optim states can be used when using Kahan Summation or Stochastic rounding (introduced in Revisiting BFloat16 Training).

For an example of the latter see: AnyPrecision optimizer with the latest version found here.

Changing precision post training

Sometimes it's OK to change precision after the model was trained.

- Using bf16-pretrained model in fp16 regime usually fails - due to overflows (the biggest number that can be represented in fp16 is 64k) for an indepth discussion and possible workaround see this PR.
- Using fp16-pretrained model in bf16 regime usually works - it will lose some performance on conversion, but should work - best to finetune a bit before using it.

Emulate a Multi-Node Setup Using Just a Single Node

The goal is to emulate a 2-node environment using a single node with 2 GPUs (for testing purposes). This, of course, can be further expanded to larger set ups.

We use the `deepspeed` launcher here. There is no need to actually use any of the deepspeed code, it's just easier to use its more advanced capabilities. You will just need to install `pip install deepspeed`.

The full setup instructions follow:

1. Create a `hostfile`:

```
$ cat hostfile
worker-0 slots=1
worker-1 slots=1
```

2. Add a matching config to your ssh client

```
$ cat ~/.ssh/config
[...]

Host worker-0
    HostName localhost
    Port 22
Host worker-1
    HostName localhost
    Port 22
```

Adapt the port if it's not 22 and the hostname if `localhost` isn't it.

3. As your local setup is probably password protected ensure to add your public key to `~/.ssh/authorized_keys`

The `deepspeed` launcher explicitly uses no-password connection, e.g. on worker0 it'd run: `ssh -o PasswordAuthentication=no worker-0 hostname`, so you can always debug ssh setup using:

```
$ ssh -vvv -o PasswordAuthentication=no worker-0 hostname
```

4. Create a test script to check both GPUs are used.

```
$ cat test1.py
import os
import time
import torch
import deepspeed
```

```
import torch.distributed as dist

# critical hack to use the 2nd gpu (otherwise both processes will use gpu0)
if os.environ["RANK"] == "1":
    os.environ["CUDA_VISIBLE_DEVICES"] = "1"

dist.init_process_group("nccl")
local_rank = int(os.environ.get("LOCAL_RANK"))
print(f'{dist.get_rank()=}, {local_rank=}')

x = torch.ones(2**30, device=f"cuda:{local_rank}")
print(x)
time.sleep(100)
```

Run:

```
$ deepspeed -H hostfile test1.py
[2022-09-08 12:02:15,192] [INFO] [runner.py:415:main] Using IP address of 192.168.0.17 for node
worker-0
[2022-09-08 12:02:15,192] [INFO] [multinode_runner.py:65:get_cmd] Running on the following
workers: worker-0,worker-1
[2022-09-08 12:02:15,192] [INFO] [runner.py:504:main] cmd = pdsh -S -f 1024 -w worker-0,worker-1
export PYTHONPATH=/mnt/nvme0/code/huggingface/multi-node-emulate-ds;  cd /mnt/nvme0/code/
huggingface/multi-node-emulate-ds; /home/stas/anaconda3/envs/py38-pt112/bin/python -u -m
deepspeed.launcher.launch --world_info=eyJ3b3JrZXItMCI6IFswXSwgIndvcmtlci0xIjogWzBdfQ==
--node_rank=%n --master_addr=192.168.0.17 --master_port=29500 test1.py
worker-0: [2022-09-08 12:02:16,517] [INFO] [launch.py:136:main] WORLD INFO DICT: {'worker-0':
[0], 'worker-1': [0]}
worker-0: [2022-09-08 12:02:16,517] [INFO] [launch.py:142:main] nnodes=2, num_local_procs=1,
node_rank=0
worker-0: [2022-09-08 12:02:16,517] [INFO] [launch.py:155:main]
global_rank_mapping=defaultdict(<class 'list'>, {'worker-0': [0], 'worker-1': [1]})
worker-0: [2022-09-08 12:02:16,517] [INFO] [launch.py:156:main] dist_world_size=2
worker-0: [2022-09-08 12:02:16,517] [INFO] [launch.py:158:main] Setting CUDA_VISIBLE_DEVICES=0
worker-1: [2022-09-08 12:02:16,518] [INFO] [launch.py:136:main] WORLD INFO DICT: {'worker-0':
[0], 'worker-1': [0]}
worker-1: [2022-09-08 12:02:16,518] [INFO] [launch.py:142:main] nnodes=2, num_local_procs=1,
node_rank=1
worker-1: [2022-09-08 12:02:16,518] [INFO] [launch.py:155:main]
global_rank_mapping=defaultdict(<class 'list'>, {'worker-0': [0], 'worker-1': [1]})
worker-1: [2022-09-08 12:02:16,518] [INFO] [launch.py:156:main] dist_world_size=2
worker-1: [2022-09-08 12:02:16,518] [INFO] [launch.py:158:main] Setting CUDA_VISIBLE_DEVICES=0
worker-1: dist.get_rank()=1, local_rank=0
worker-0: dist.get_rank()=0, local_rank=0
worker-1: tensor([1., 1., 1.,  ..., 1., 1., 1.], device='cuda:0')
worker-0: tensor([1., 1., 1.,  ..., 1., 1., 1.], device='cuda:0')
```

If the ssh set up works you can run `nvidia-smi` in parallel and observe that both GPUs allocated ~4GiB of memory from `torch.ones` call.

Note that the script hacks in `CUDA_VISIBLE_DEVICES` to tell the 2nd process to use gpu1, but it'll be seen as `local_rank==0` in both cases.

5. Finally, let's test that NCCL collectives work as well

Script adapted from torch-distributed-gpu-test.py to just tweak `os.environ["CUDA_VISIBLE_DEVICES"]`

```
$ cat test2.py
import deepspeed
import fcntl
import os
import socket
import time
import torch
import torch.distributed as dist

# a critical hack to use the 2nd GPU by the 2nd process (otherwise both processes will use gpu0)
if os.environ["RANK"] == "1":
    os.environ["CUDA_VISIBLE_DEVICES"] = "1"

def printflock(*msgs):
    """ solves multi-process interleaved print problem """
    with open(__file__, "r") as fh:
        fcntl.flock(fh, fcntl.LOCK_EX)
        try:
            print(*msgs)
        finally:
            fcntl.flock(fh, fcntl.LOCK_UN)

local_rank = int(os.environ["LOCAL_RANK"])
torch.cuda.set_device(local_rank)
device = torch.device("cuda", local_rank)
hostname = socket.gethostname()

gpu = f"[{hostname}-{local_rank}]"

try:
    # test distributed
    dist.init_process_group("nccl")
    dist.all_reduce(torch.ones(1).to(device), op=dist.ReduceOp.SUM)
    dist.barrier()
    print(f'{dist.get_rank()=}, {local_rank=}')

    # test cuda is available and can allocate memory
    torch.cuda.is_available()
    torch.ones(1).cuda(local_rank)

    # global rank
    rank = dist.get_rank()
    world_size = dist.get_world_size()
```

```
    printflock(f"{gpu} is OK (global rank: {rank}/{world_size})")

    dist.barrier()
    if rank == 0:
        printflock(f"pt={torch.__version__}, cuda={torch.version.cuda},
nccl={torch.cuda.nccl.version()}")
        printflock(f"device compute capabilities={torch.cuda.get_device_capability()}")
        printflock(f"pytorch compute capabilities={torch.cuda.get_arch_list()}")

except Exception:
    printflock(f"{gpu} is broken")
    raise
```

Run:

```
$ deepspeed -H hostfile test2.py
[2022-09-08 12:07:09,336] [INFO] [runner.py:415:main] Using IP address of 192.168.0.17 for node
worker-0
[2022-09-08 12:07:09,337] [INFO] [multinode_runner.py:65:get_cmd] Running on the following
workers: worker-0,worker-1
[2022-09-08 12:07:09,337] [INFO] [runner.py:504:main] cmd = pdsh -S -f 1024 -w worker-0,worker-1
export PYTHONPATH=/mnt/nvme0/code/huggingface/multi-node-emulate-ds;  cd /mnt/nvme0/code/
huggingface/multi-node-emulate-ds; /home/stas/anaconda3/envs/py38-pt112/bin/python -u -m
deepspeed.launcher.launch --world_info=eyJ3b3JrZXItMCI6IFswXSwgIndvcmtlci0xIjogWzBdfQ==
--node_rank=%n --master_addr=192.168.0.17 --master_port=29500 test2.py
worker-0: [2022-09-08 12:07:10,635] [INFO] [launch.py:136:main] WORLD INFO DICT: {'worker-0':
[0], 'worker-1': [0]}
worker-0: [2022-09-08 12:07:10,635] [INFO] [launch.py:142:main] nnodes=2, num_local_procs=1,
node_rank=0
worker-0: [2022-09-08 12:07:10,635] [INFO] [launch.py:155:main]
global_rank_mapping=defaultdict(<class 'list'>, {'worker-0': [0], 'worker-1': [1]})
worker-0: [2022-09-08 12:07:10,635] [INFO] [launch.py:156:main] dist_world_size=2
worker-0: [2022-09-08 12:07:10,635] [INFO] [launch.py:158:main] Setting CUDA_VISIBLE_DEVICES=0
worker-1: [2022-09-08 12:07:10,635] [INFO] [launch.py:136:main] WORLD INFO DICT: {'worker-0':
[0], 'worker-1': [0]}
worker-1: [2022-09-08 12:07:10,635] [INFO] [launch.py:142:main] nnodes=2, num_local_procs=1,
node_rank=1
worker-1: [2022-09-08 12:07:10,635] [INFO] [launch.py:155:main]
global_rank_mapping=defaultdict(<class 'list'>, {'worker-0': [0], 'worker-1': [1]})
worker-1: [2022-09-08 12:07:10,635] [INFO] [launch.py:156:main] dist_world_size=2
worker-1: [2022-09-08 12:07:10,635] [INFO] [launch.py:158:main] Setting CUDA_VISIBLE_DEVICES=0
worker-0: dist.get_rank()=0, local_rank=0
worker-1: dist.get_rank()=1, local_rank=0
worker-0: [hope-0] is OK (global rank: 0/2)
worker-1: [hope-0] is OK (global rank: 1/2)
worker-0: pt=1.12.1+cu116, cuda=11.6, nccl=(2, 10, 3)
worker-0: device compute capabilities=(8, 0)
worker-0: pytorch compute capabilities=['sm_37', 'sm_50', 'sm_60', 'sm_70', 'sm_75', 'sm_80',
'sm_86']
worker-1: [2022-09-08 12:07:13,642] [INFO] [launch.py:318:main] Process 576485 exits
```

```
successfully.
worker-0: [2022-09-08 12:07:13,642] [INFO] [launch.py:318:main] Process 576484 exits
successfully.
```

Voila, mission accomplished.

We tested that the NCCL collectives work, but they use local NVLink/PCIe and not the IB/ETH connections like in real multi-node, so it may or may not be good enough for testing depending on what needs to be tested.

Larger set ups

Now, let's say you have 4 GPUs and you want to emulate 2x2 nodes. Then simply change the `hostfile` to be:

```
$ cat hostfile
worker-0 slots=2
worker-1 slots=2
```

and the `CUDA_VISIBLE_DEVICES` hack to:

```
if os.environ["RANK"] in ["2", "3"]:
    os.environ["CUDA_VISIBLE_DEVICES"] = "2,3"
```

Everything else should be the same.

Automating the process

If you want an automatic approach to handle any shape of topology, you could use something like this:

```
def set_cuda_visible_devices():
    """
    automatically assign the correct groups of gpus for each emulated node by tweaking the
    CUDA_VISIBLE_DEVICES env var
    """

    global_rank = int(os.environ["RANK"])
    world_size = int(os.environ["WORLD_SIZE"])
    emulated_node_size = int(os.environ["LOCAL_SIZE"])
    emulated_node_rank = int(global_rank // emulated_node_size)
    gpus = list(map(str, range(world_size)))
    emulated_node_gpus =
","".join(gpus[emulated_node_rank*emulated_node_size:(emulated_node_rank+1)*emulated_node_size])
    print(f"Setting CUDA_VISIBLE_DEVICES={emulated_node_gpus}")
    os.environ["CUDA_VISIBLE_DEVICES"] = emulated_node_gpus

set_cuda_visible_devices()
```

Emulating multiple GPUs with a single GPU

The following is an orthogonal need to the one discussed in this document, but it's related so I thought it'd be useful to share some insights here:

With NVIDIA A100 you can use MIG to emulate up to 7 instances of GPUs on just one real GPU, but for years you couldn't use those instances for anything but standalone use - e.g. you couldn't do DDP or any NCCL comms over those GPUs. I hoped I could use my A100 to emulate 7 instances and add one more real GPU and to have 8x GPUs to do development with - but nope it didn't work. Asking NVIDIA engineers about it at the time, there were no plans to have this use-case supported.

That has changed - NCCL 2.31 (2026) added experimental support for communicators that span MIG instances, so a single physical GPU can now stand in for a small multi-GPU node and you can run DDP, FSDP, ZeRO and any NCCL collective across its instances. What follows is a working recipe on an H200, and measurements of how fast those emulated GPUs actually talk to each other.

Getting a new enough NCCL

You need an NVIDIA driver with CUDA 13.0 or newer, and NCCL 2.31 or newer.

As of Aug 2026, Pytorch's latest 2.13.0 bundles a lower version. Until a PyTorch build arrives with NCCL>=2.31 in it, install NCCL yourself and load it ahead of the bundled one:

```
pip install -U --target $HOME/nccl-new 'nvidia-nccl-cu13>=2.31'
LD_PRELOAD=$HOME/nccl-new/nvidia/nccl/lib/libnccl.so.2 python train.py
```

It goes into its own directory rather than into the environment because PyTorch pins an exact NCCL version (`nvidia-nccl-cu13==2.29.7` for 2.13.0) and installing over it would put pip's dependency resolver at odds with torch's metadata. `LD_PRELOAD` then makes the dynamic linker resolve NCCL's symbols from the new library before it reaches the one that ships inside the torch wheel.

Just to make sure run the program once with `NCCL_DEBUG=INFO` and check that the banner says `NCCL version 2.31.2+cuda13.3` or newer. Don't use `torch.cuda.nccl.version()` for this check - it reports the version PyTorch was built against, not the preloaded one that is actually doing the work.

The NCCL settings MIG needs

```
export NCCL_MULTI_RANK_GPU_ENABLE=1
export NCCL_NVLS_ENABLE=0
```

The NCCL docs say `NCCL_MULTI_RANK_GPU_ENABLE` isn't required, since each MIG instance is a distinct CUDA device - but on an H200 with NCCL 2.31.2 the communicator won't come up without it, so set it anyway. NCCL identifies a rank's device by the parent GPU's NVML index and PCI bus id rather than by the MIG instance, so four ranks on four instances of one GPU look to it like four ranks sharing a single device, and without the flag init stops with `Multiple Ranks are using the same GPU/Partition`.

`NCCL_NVLS_ENABLE=0` is a guard rather than a requirement. NCCL's default already leaves NVLink SHARP off in this situation, so if nothing in your environment sets that variable you can leave it alone. But many cluster images and pod specs export `NCCL_NVLS_ENABLE=1`, and when NVLink SHARP is explicitly demanded NCCL refuses to start with `NCCL_NVLS_ENABLE has been set to "1" and communicator has multiple ranks using the same NVML device`. Setting it to 0 costs nothing and makes the recipe work regardless of what the environment already exported. You can also set `NCCL_NVLS_ENABLE=2` which will automatically use NVLS when possible.

The docs also tell you to set `NCCL_MNNVL_ENABLE=0`. On a node with no NVLink fabric that flag changes nothing - the same benchmark returned 142.28, 142.42 and 142.23 GBps with it unset, set to `0` and set to `1` - so it only matters on fabric-attached systems like GB200 NVL72.

Carving up a GPU

List the profiles the GPU offers, enable MIG on it, then create the instances. On H200 we can create up to 7 instances. Let's create 4 instances of 35GB each on GPU 4 which will allows us to do for example SP=2, EP=2 model parallelism on a single GPU.

```
nvidia-smi mig -lgip             # show profiles available on this GPU
sudo nvidia-smi -i 4 -mig 1      # enable MIG on GPU 4
sudo nvidia-smi mig -i 4 -cgi 1g.35gb,1g.35gb,1g.35gb,1g.35gb -C
```

To undo MIG and to regain the full GPU:

```
sudo nvidia-smi mig -i 4 -dci && sudo nvidia-smi mig -i 4 -dgi
sudo nvidia-smi -i 4 -mig 0
```

Finding the MIG UUIDs

Each rank must be given exactly one instance through `CUDA_VISIBLE_DEVICES`, and MIG instances are addressed by UUID rather than by index. `nvidia-smi -L` prints them indented under their parent GPU:

```
$ nvidia-smi -L
[...]
GPU 4: NVIDIA H200 (UUID: GPU-31f70a1b-dee6-617e-13c5-5c2a2fbe0013)
  MIG 1g.35gb  Device 0: (UUID: MIG-e26b1e11-8424-5ef0-97a8-d87e519f7824)
  MIG 1g.35gb  Device 1: (UUID: MIG-1a7d7808-5dcc-50fa-842c-2ceaa9f6e8ee)
  MIG 1g.35gb  Device 2: (UUID: MIG-46c30315-dfa5-56d4-82f6-345f84b4d52d)
  MIG 1g.35gb  Device 3: (UUID: MIG-6b4ff05d-689c-5bf3-8a51-3ff519c617ac)
```

Collect every rank-sized device on the node into one comma-separated variable - a GPU that is in MIG mode contributes its instances, a GPU that isn't contributes itself:

```
export MIG_UUIDS=$(nvidia-smi -L | perl -ne 'if (/\(UUID: (GPU-[^)]+)\)/) { push @u, $1 } elsif
(/\(UUID: (MIG-[^)]+)\)/) { my $m = $1; pop @u if $u[-1] =~ /^GPU-/; push @u, $m } END { print
join ",", @u }')
```

Launching the ranks

`torchrun` hands each process a `LOCAL_RANK`, which is all that's needed to give each one its own instance:

```
torchrun --nproc_per_node 4 --no-python bash -c '
export CUDA_VISIBLE_DEVICES=$(echo $MIG_UUIDS | cut -d, -f$((LOCAL_RANK+1)))
```

```
export LOCAL_RANK=0
exec python train.py'
```

Resetting `LOCAL_RANK` to `0` is required, see Caveats.

`all_reduce`, `all_gather`, `reduce_scatter`, `broadcast` and a DDP training step all work across 4 such instances.

How fast do the emulated GPUs talk to each other?

The emulation is only useful if the collectives aren't crawling, so here is all_reduce_bench.py run on MIG instances of one H200, measured against the same number of whole H200s - 4 instances against 4 real GPUs, 2 instances against 2 real GPUs.

The profiles used below, all on one H200 141GB:

MIG profile	compute slices	memory	instances per GPU
`1g.18gb`	1 of 7	18GB	7
`1g.35gb`	1 of 7	35GB	4
`2g.35gb`	2 of 7	35GB	3
`3g.71gb`	3 of 7	71GB	2

Every results table below starts with the whole-GPU row it is measured against, then the MIG rows in descending `busbw`.

4 ranks, 512MiB payload:

what the 4 ranks run on	busbw	vs whole GPUs
4x whole H200 (NVLink)	341.1GBps	100%
4x `1g.35gb` of one GPU	273.8GBps	80%
4x `1g.18gb` of one GPU	142.3GBps	42%

4 ranks, 2GiB payload:

what the 4 ranks run on	busbw	vs whole GPUs
4x whole H200 (NVLink)	354.7GBps	100%
4x `1g.35gb` of one GPU	281.3GBps	79%
4x `1g.18gb` of one GPU	144.7GBps	41%

2 ranks, 512MiB payload:

what the 2 ranks run on	busbw	vs whole GPUs
2x whole H200 (NVLink)	308.2GBps	100%
2x `3g.71gb` of one GPU	413.8GBps	134%

what the 2 ranks run on	busbw	vs whole GPUs
2x `2g.35gb` of one GPU	245.3GBps	80%
2x `1g.35gb` of one GPU	230.9GBps	75%
2x `1g.18gb` of one GPU	122.0GBps	40%

2 ranks, 2GiB payload:

what the 2 ranks run on	busbw	vs whole GPUs
2x whole H200 (NVLink)	331.7GBps	100%
2x `3g.71gb` of one GPU	429.5GBps	129%
2x `2g.35gb` of one GPU	250.4GBps	75%
2x `1g.35gb` of one GPU	235.0GBps	71%
2x `1g.18gb` of one GPU	123.6GBps	37%

In the profile table above, the compute-slice count is the digit in the profile name (`2g.35gb` gets 2 of the GPU's 7 compute slices) and the memory figure says how many of its 8 memory slices the instance owns - 18GB is 1 slice, 35GB is 2, 71GB is 4.

Setup: H200 141GB SXM, `torch==2.9.1+cu130`, NCCL 2.31.2 preloaded as above, CUDA driver 13.0.

The big surprise here is how much the choice of profile matters. Traffic between MIG instances never leaves the GPU - NCCL logs every channel as `P2P/CUMEM` with no socket or PCIe fallback anywhere in the run - so the limit is the memory bandwidth the instance owns, which is set by its memory slices and barely at all by its SMs. The two 512MiB tables isolate this: in the 4-rank one, going from `1g.18gb` to `1g.35gb` keeps compute identical at one slice and doubles the memory, and bandwidth nearly doubles (142 -> 274GBps), while in the 2-rank one, going from `1g.35gb` to `2g.35gb` doubles the compute at the same memory and buys only 6% (231 -> 245GBps).

So 4x `1g.35gb` instances reproduce about 79% of the interconnect bandwidth of 4 real H200s, while 4x `1g.18gb` instances - same rank count, smaller slices - give only 41%. Pick the profile with the most memory per instance that still gives you the rank count you need.

And with only 2 ranks, `3g.71gb` instances are actually *faster* than 2 whole GPUs (429 against 332GBps in the 2-rank 2GiB table), because a copy inside one chip's HBM beats a hop over NVLink.

Spreading the instances over 2 GPUs

Until NVIDIA overcomes the odd 7-instance per GPU limit, if you want to do a 3D parallelism, you need at least 8 GPUs or instances. So for a time being we have to use 2 GPUs to get to 8.

Everything above keeps the whole emulated world inside one physical GPU. Partitioning 2 GPUs into 4 instances each gives 8 ranks, and the setup plus the benchmark is:

```
for i in 0 1; do
    sudo nvidia-smi -i $i -mig 1
    sudo nvidia-smi mig -i $i -cgi 1g.35gb,1g.35gb,1g.35gb,1g.35gb -C
done
```

```
export MIG_UUIDS=$(nvidia-smi -L | perl -ne 'if (/\(UUID: (GPU-[^)]+)\)/) { push @u, $1 } elsif
(/\(UUID: (MIG-[^)]+)\)/) { my $m = $1; pop @u if $u[-1] =~ /^GPU-/; push @u, $m } END { print
join ",", @u }')

torchrun --nproc_per_node 8 --no-python bash -c '
export CUDA_VISIBLE_DEVICES=$(echo $MIG_UUIDS | cut -d, -f$((LOCAL_RANK+1)))
export LOCAL_RANK=0
exec python all_reduce_bench.py --payload_size_in_gib 2'
```

8 ranks, 512MiB payload:

what the 8 ranks run on	busbw	vs whole GPUs
8x whole H200 (NVLink)	450.2GBps	100%
4x `1g.35gb` on each of 2 GPUs	26.8GBps	6%

8 ranks, 2GiB payload:

what the 8 ranks run on	busbw	vs whole GPUs
8x whole H200 (NVLink)	465.9GBps	100%
4x `1g.35gb` on each of 2 GPUs	25.6GBps	5%

That is a collapse, and it is not a small-message effect - the 2GiB payload is no better than the 512MiB one. NCCL's own log says why: of the ring's channels, 24 hops went `via P2P/CUMEM` and 6 went `via SHM/direct`. The hops that stay inside a GPU keep the fast path, but the ones that cross to the other GPU fall back to host shared memory, because the CUDA driver doesn't support NVLink P2P for MIG instances - the same limitation that rules out NVLink SHARP here. A ring runs at the speed of its slowest link, so those few host-memory hops set the pace for all 8 ranks.

Mixing MIG instances and full GPUs

Not every GPU has to be partitioned. Since 7 instances is the most one GPU will give you, the cheapest route to an 8-rank world is to carve one H200 (or another high end GPU) into its maximum 7 instances and leave a second GPU whole. The `MIG_UUIDS` command from above already produces the right list for that mix - the partitioned GPU contributes its 7 instances, the whole one contributes itself:

```
sudo nvidia-smi -i 0 -mig 1
sudo nvidia-smi mig -i 0 -cgi 1g.18gb,1g.18gb,1g.18gb,1g.18gb,1g.18gb,1g.18gb,1g.18gb -C

export MIG_UUIDS=$(nvidia-smi -L | perl -ne 'if (/\(UUID: (GPU-[^)]+)\)/) { push @u, $1 } elsif
(/\(UUID: (MIG-[^)]+)\)/) { my $m = $1; pop @u if $u[-1] =~ /^GPU-/; push @u, $m } END { print
join ",", @u }')

torchrun --nproc_per_node 8 --no-python bash -c '
export CUDA_VISIBLE_DEVICES=$(echo $MIG_UUIDS | cut -d, -f$((LOCAL_RANK+1)))
export LOCAL_RANK=0
exec python all_reduce_bench.py --payload_size_in_gib 2'
```

8 ranks, 512MiB payload:

what the 8 ranks run on	busbw	vs whole GPUs
8x whole H200 (NVLink)	450.2GBps	100%
7x `1g.18gb` of one GPU + 1 whole H200	28.5GBps	6%

8 ranks, 2GiB payload:

what the 8 ranks run on	busbw	vs whole GPUs
8x whole H200 (NVLink)	465.9GBps	100%
7x `1g.18gb` of one GPU + 1 whole H200	27.5GBps	6%

The whole GPU rescues nothing: 24 hops ran `via P2P/CUMEM` inside the partitioned GPU and 6 `via SHM/direct` to the whole one, and the ring again runs at the speed of the host-memory hops. The useful corollary is that the second device's own bandwidth doesn't matter - a whole H200 in the eighth slot gave 28.5GBps, no better than the 26.8GBps of the run where that eighth rank was a MIG instance on a second partitioned H200. So if the goal is only to get past the 7-instances-per-GPU ceiling, the second device can be any GPU CUDA will talk to, and it needs no MIG support of its own since it joins whole - one H200 plus one modest GPU is an 8-rank world.

The practical rule: keep an emulated multi-GPU job inside a single physical GPU. Crossing to a second GPU costs a factor of 10 against 4 instances on one GPU (26.8 against 273.8GBps at 512MiB), and leaves you at 5% of what 8 real GPUs do. Though if you're working with a tiny model and payload it probably doesn't matter much for dev purposes.

Do keep in mind that this emulates the *topology* of a multi-GPU job and not its performance - each rank also gets only a fraction of the SMs and a fraction of the memory, so it's a development and testing tool, not a way to measure throughput.

Caveats

`LOCAL_RANK` is `0` in every rank

Once `CUDA_VISIBLE_DEVICES` names a single instance, the process sees exactly one device and CUDA numbers it `0`, so code doing `torch.cuda.set_device(local_rank)` with the value torchrun handed out would fail with `invalid device ordinal`. There is no way around it: CUDA enumerates only a single compute instance per process even when several MIG devices are visible. It stays true when the ranks span 2 partitioned GPUs - which physical GPU a rank landed on is carried by the UUID, not by the ordinal, so every rank says `cuda:0` and every rank is on different hardware.

That leaves every rank reporting `LOCAL_RANK=0`, which is worth being precise about.

Nothing in `torch.distributed` is affected. `dist.get_rank()`, `dist.get_world_size()` and the process group are built from `RANK` and `WORLD_SIZE`, which the launch wrapper doesn't touch. The usual device-selection idiom also keeps working unchanged:

```
local_rank = int(os.environ["LOCAL_RANK"])   # 0 in every process
torch.cuda.set_device(local_rank)            # binds to this rank's own instance
model = DDP(model.to(local_rank), device_ids=[local_rank])
```

Each process asks for device 0 and each one gets a different slice, because `CUDA_VISIBLE_DEVICES` differs per rank.

What breaks is code that treats `LOCAL_RANK` as an identity rather than as a device ordinal, and the node-leader guard is the common case:

```
if local_rank == 0:
    download_dataset()   # meant to run once per node
```

Under MIG that condition is true in every process, so all the ranks download the dataset, write the same log file, or race to populate the same cache directory.

In a single-node emulation the fix is to use `RANK`, which is still unique across the job:

```
if dist.get_rank() == 0:
    download_dataset()
```

If the code must keep working unmodified on real multi-node runs, preserve the original slot number under a different name instead. The launch wrapper stashes it before overwriting:

```
torchrun --nproc_per_node 8 --no-python bash -c '
export MIG_SLOT=$LOCAL_RANK
export CUDA_VISIBLE_DEVICES=$(echo $MIG_UUIDS | cut -d, -f$((MIG_SLOT+1)))
export LOCAL_RANK=0
exec python train.py'
```

and the guard reads whichever variable is present:

```
node_slot = int(os.environ.get("MIG_SLOT", os.environ["LOCAL_RANK"]))
if node_slot == 0:
    download_dataset()
```

Only 7 instances per GPU

MIG cuts along boundaries the silicon already has. A compute slice comes from a GPC (Graphics Processing Cluster), the top-level block that bundles a set of SMs with its own work distribution, and a memory slice comes from the memory system attached to the HBM stacks. Ampere, Hopper, Blackwell, and recently Rubin, according to NVIDIA sources, each contain exactly 8 GPCs in the full GPU implementation (though a shipping SKU may have fewer enabled).

MIG can have a maximum of 7 instances and not 8 because the eighth compute unit is reserved to manage the instances - it is held back the moment MIG mode is enabled, whether you then create 7 instances or one. The memory system has no such tenant, so all 8 memory slices stay available, and the MIG documentation states the resulting ratio outright: each memory slice is 1/8 of the memory, each SM slice is 1/7 of the SMs.

That asymmetry is what the profile table above shows. Seven `1g` instances consume all 7 compute slices but only 7 of the 8 memory slices, so one slice worth of HBM is stranded on a fully partitioned GPU - 18GB of the H200's 141GB. It is also why a `4g` profile allows just one instance: a second would need 8 compute slices and there are only 7.

`torch.cuda.device_count()` disagrees with `LOCAL_WORLD_SIZE`

A process can see only its own instance, so `torch.cuda.device_count()` returns 1 in every rank. On whole GPUs that call

happens to equal the number of ranks on the node, and plenty of code relies on it - to size a shard count, to build a list of devices to iterate, or to check that the requested world fits on the node. Take the number from the environment instead:

```
ranks_on_this_node = torch.cuda.device_count()               # 1 in every rank under MIG
ranks_on_this_node = int(os.environ["LOCAL_WORLD_SIZE"])     # what torchrun was given as
--nproc_per_node
```

`LOCAL_WORLD_SIZE` is set by torchrun in every worker, alongside `RANK`, `LOCAL_RANK` and `WORLD_SIZE`, and it keeps reporting the full local rank count under MIG.

Re-Train HF Hub Models From Scratch Using Finetuning Examples

HF Transformers has awesome finetuning examples https://github.com/huggingface/transformers/tree/main/examples/pytorch, that cover pretty much any modality and these examples work out of box.

But what if you wanted to re-train from scratch rather than finetune.

Here is a simple hack to accomplish that.

We will use `facebook/opt-1.3b` and we will plan to use bf16 training regime as an example here:

```
cat << EOT > prep-bf16.py
from transformers import AutoConfig, AutoModelForCausalLM, AutoTokenizer
import torch

mname = "facebook/opt-1.3b"

config = AutoConfig.from_pretrained(mname)
model = AutoModelForCausalLM.from_config(config, torch_dtype=torch.bfloat16)
tokenizer = AutoTokenizer.from_pretrained(mname)

path = "opt-1.3b-bf16"

model.save_pretrained(path)
tokenizer.save_pretrained(path)
EOT
```

now run:

```
python prep-bf16.py
```

This will create a folder: `opt-1.3b-bf16` with everything you need to train the model from scratch. In other words you have a pretrained-like model, except it only had its initializations done and none of the training yet.

Adjust the script above to use `torch.float16` or `torch.float32` if that's what you plan to use instead.

Now you can proceed with finetuning this saved model as normal:

```
torchrun \
--nproc_per_node=1 --nnode=1 --node_rank=0 \
--master_addr=127.0.0.1 --master_port=9901 \
examples/pytorch/language-modeling/run_clm.py --bf16 \
--seed 42 --model_name_or_path opt-1.3b-bf16 \
--dataset_name wikitext --dataset_config_name wikitext-103-raw-v1 \
--per_device_train_batch_size 12 --per_device_eval_batch_size 12 \
```

```
--gradient_accumulation_steps 1 --do_train --do_eval --logging_steps 10 \
--save_steps 1000 --eval_steps 100 --weight_decay 0.1 --num_train_epochs 1 \
--adam_beta1 0.9 --adam_beta2 0.95 --learning_rate 0.0002 --lr_scheduler_type \
linear --warmup_steps 500 --report_to tensorboard --output_dir save_dir
```

The key entry being:

```
--model_name_or_path opt-1.3b-bf16
```

where `opt-1.3b-bf16` is your local directory you have just generated in the previous step.

Sometimes it's possible to find the same dataset that the original model was trained on, sometimes you have to use an alternative dataset.

The rest of the hyper-parameters can often be found in the paper or documentation that came with the model.

To summarize, this recipe allows you to use finetuning examples to re-train whatever model you can find on the HF hub.

Dealing with Datasets

Preprocessing and caching datasets on the main process

HF Accelerate has a very neat container `main_process_first` which allows to write code like:

```
with accelerator.main_process_first():
    # load and pre-process datasets
    dataset = datasets.load_dataset(...)
    # optionally cache it and have the rest of the processes load the cache
```

instead of the less intuitive and requiring code repetition:

```
if rank == 0:
    dataset = datasets.load_dataset(...)
dist.barrier()
if not rank == 0:
    dataset = datasets.load_dataset(...)
```

You want to download and process data on the main process and not all processes, because they will be all repeating the same thing in parallel and more over are likely to write to the same location which will result in interleaved broken result. It's also much faster IO-wise to serialize such work.

Now there is `main_process_first` and `local_main_process_first` - the first one is for when your data resides on a shared filesystem and all compute nodes can see it. The second one is for when the data is local to each node.

If you aren't using HF Accelerate, I have recreated similar containers, except called them:

- `global_main_process_first` - for shared fs
- `local_main_process_first` - for local to node fs

You can find them here.

Now, what if you want to write a generic code that automatically works on shared and local filesystems. I added another helper that automatically discovers what type of filesystem we are dealing with and based on that call the right containers. I called it `main_process_by_path_first`, which is used like:

```
path = "/path/to/data"
with main_process_by_path_first(path):
    # load and pre-process datasets
    dataset = datasets.load_dataset(...)
    # optionally cache it and have the rest of the processes load the cache
```

You can find it here.

Of course, besides containers you will also want utils to check the type of main process, and so there are 3 of those corresponding to the containers:

- `is_main_process_by_path(path)`
- `is_local_main_process()`
- `is_global_main_process()`

They are all found in here.

You can see them in action by running:

```
python -u -m torch.distributed.run --nproc_per_node=2 --rdzv_endpoint localhost:6000
--rdzv_backend c10d tools/main_process_first.py
```

Inference

XXX: this chapter is under construction - some sections are complete, some are still starting out, many are yet to be started, but there are already enough of useful sections completed to make it a good reading.

Glossary

- CLA: Cross-Layer Attention
- FHE: Fully Homomorphic Encryption
- GQA: Grouped-Query Attention
- ITL: Inter-Token Latency
- KV: Key Value
- LPU: Language Processing Unit™
- MHA: Multi-Head Attention
- MLA: Multi-Latent Attention
- MPC: Secure Multi-Party Computation
- MQA: Multi-Query Attention
- PPML: Privacy-Preserving Machine Learning
- QPS: Queries Per Second
- TPOT: Time Per Output Token
- TTFT: Time to First Token

See Concepts for more glossary-like entries.

Concepts

Prefill and Decode

When doing inference there are 2 stages:

Prefill

During prefill, the model processes the prompt tokens in parallel, similar to training, and builds the KV cache used during generation. Prefill is usually compute-bound. Its latency contributes directly to TTFT and generally increases with input length; long prompts or insufficient compute can make prefill a major part of request latency.

Decode

During ordinary autoregressive decoding, the model generates one output token at a time based on the prompt and previously generated tokens. This sequential dependency prevents parallelizing output-token steps within one sequence, and decode is often memory-bandwidth-bound.

Which stage matters most depends on the workload and metric. TTFT is sensitive to prompt length, prefill efficiency, scheduling, and queueing; TPOT/ITL primarily characterizes decode; end-to-end latency depends on input length, output length, batching, load, and hardware.

See NVIDIA's inference optimization overview and LLM benchmarking metric definitions.

Anatomy of Model's Memory Usage

The inference memory usage is quite different from training. Here we have:

1. Model weights
2. KV cache - crucial to not need to recalculate past tokens for each new generated token
3. Activation memory - this is the processing temporary memory which would depend on a batch size and a sequence length

Model Weights

Bytes per parameter for each dtype:

- 4 bytes * number of parameters for fp32
- 2 bytes * number of parameters for fp16/bf16
- 1 byte * number of parameters for fp8/int8
- 0.75 bytes * number of parameters for fp6 or 6-bit quantization
- 0.625 bytes * number of parameters for 5-bit quantization
- 0.5 bytes * number of parameters for fp4/int4
- 0.375 bytes * number of parameters for 3-bit quantization
- 0.25 bytes * number of parameters for 2-bit quantization
- 0.125 bytes * number of parameters for 1-bit quantization

These are densely packed payload sizes. Actual weight storage can be higher because quantized formats may store scales, zero points, codebooks, padding/alignment, and selected tensors in a wider format.

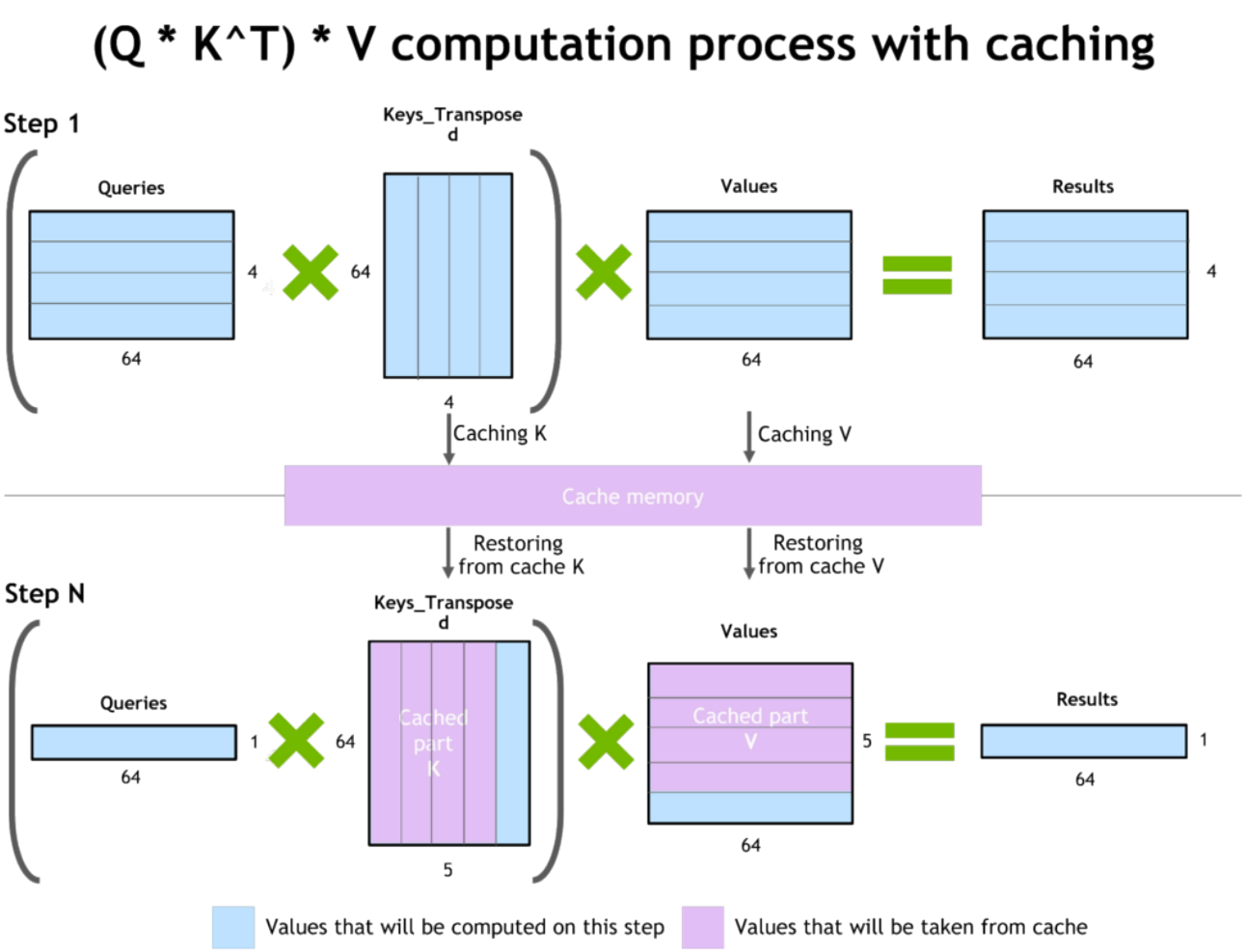
The OCP microscaling (MX) formats use one 8-bit E8M0 scale per block of 32 values. Including the shared scale, their packed payload sizes are:

- 1.03125 bytes * number of parameters for MXFP8/MXINT8
- 0.78125 bytes * number of parameters for MXFP6
- 0.53125 bytes * number of parameters for MXFP4

Example: Meta-Llama-3.1-8B in bf16 will need `2 (bf16 bytes) * 8B (num of params) = 16GB` (approximately)

KV Caching

It'd be very expensive to recalculate all the previous KV (Key Value) values before each new token is generated and thus they are cached in accelerator's memory. Newly computed KV-values are appended to the existing cache.



(source)

KV cache size is directly proportional to the input sequence length and batch size. Past query values aren't used in the attention mechanism and thus don't need to be cached.

A KV cache of 1 token requires `dtype_bytes * 2 * num_hidden_layers * hidden_size * num_key_value_heads / num_attention_heads` bytes

notes:

- `dtype_bytes` is bytes per dtype: 4 bytes for fp32, 2 bytes for bf16/fp16, etc.
- 2 stands for keys + values as there are 2 of them.
- `num_key_value_heads / num_attention_heads` is the factor that will depend on whether multi-query (MQA), grouped-query (GQA) or multi-head attention (MHA) is used. for MHA it'll be 1, for MQA it'll be `1/num_attention_heads` and for GQA it'll depend on how many queries are used per group, i.e. `num_key_value_heads / num_attention_heads` which is the general case for MHA and MQA.

You can get these dimensions from `config.json` inside the model's folder or from an equivalent file if it's different. e.g. meta-llama/Meta-Llama-3.1-8B.

Examples:

1 token Meta-Llama-3.1-8B in bf16 will need: `2 (bf16 bytes) * 2 (keys+values) * 32 (num_hidden_layers) * 4096 (hidden_size) * 8 (num_key_value_heads) / 32 (num_attention_heads) / 10**6 = 0.131MB`. This model uses GQA so it uses 1/4th of the vanilla MHA.

A batch size of 1 of 1024 tokens will need `0.131*1024 = ~134MB`.

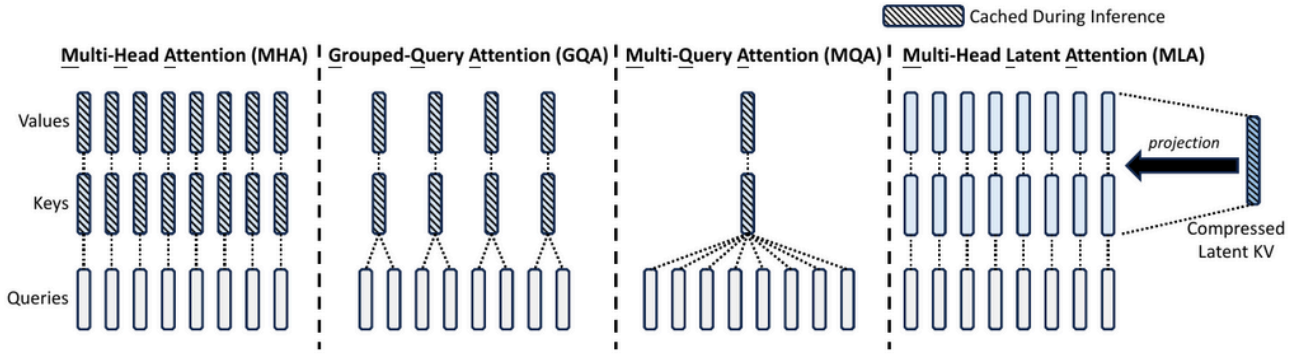
A batch size of 128 of 1024 tokens each will need `0.131*1024*128 / 10**3 = ~17.2GB`.

The KV cache for Meta-Llama-3.1-8B would have taken 4x more memory per token if it were to use MHA, 8x less memory if it were to use MQA. It's easy to see why from the MHA/GQA/MQA/MLA diagram further below.

In this case the model has `num_key_value_heads=8` and `num_attention_heads=32`, hence MQA and GQA use 32x and 4x less memory than MHA, correspondingly.

DeepSeek v3 introduced Multi-Latent Attention (MLA) which compresses the Key and Value into a latent vector, which further reduces the KV-cache size. See section 2.1.1 of the paper for the specific details.

Here is the diagram that shows the difference between MHA/GQA/MQA/MLA:



source

SwiftKV was invented to deal with the common situation of 10:1 ratio of prefill vs decode use-cases, reducing inference computation during prompt processing rather than just compressing memory. By combining model rewiring and knowledge-preserving self-distillation, SwiftKV achieves substantial reductions in computational overhead during inference with minimal accuracy loss, leading to transformative improvements in throughput, latency and cost efficiency for enterprise LLM workloads by up to 2x.

KV cache while saving recomputation has a big negative impact on inference's performance. Here is a quote from Dynamic Memory Compression: Retrofitting LLMs for Accelerated Inference:

> 2.3. Memory-Bound and Compute-Bound Operations Every operation performed with a GPU accelerator, such as General Matrix Multiply (GEMM), is either memory-bound or compute-bound. In the former case, the overall runtime is dominated by high bandwidth memory (HBM) access, while in the latter by the actual computations. Auto-regressive generation with Transformer LLMs, where the sequence length for every forward pass is n = 1, tends to be memory-bound rather than compute-bound. The vast majority of a forward pass is spent either processing linear layers (in MHSA, Feed-Forward, and output vocabulary projection) or calculating attention scores and outputs from Equation (4). For linear layers, the ratio of FLOPS to memory accesses improves as the batch size increases, and more FLOPS are performed with the set of layer weights retrieved from the HBM. Eventually, with a large enough batch size, linear layers become compute-bound. On the other hand, for the calculation of Equation (4) inside MHSA layers during auto-regressive inference, the ratio of FLOPS to input size remains constant, and MHSA layers are memory-bound regardless of the batch size. It follows that for those layers, latency scales linearly with the size of the KV cache.

- Equation (4) is the usual self-attention mechanism equation of `Softmax(Q,K)V`

A smaller KV cache would lead to faster generation and higher GPU utilization. So various techniques like gisting, context distillation, key-value eviction policies (token dropping), memory compression, multi-query attention, grouped-query

attention, cross-layer attention, anchor-based self-attention, quantization and many others are used to accomplish that.

Online vs Offline inference

When you have users that send queries in real time - this is Online inference, also known as Deployment or Interactive inference. Examples: chatbot, search engines, general REST APIs. In this case one always runs an inference server and there could be various clients querying it.

When you have a file with hundreds or thousands of prompts that you need to run inference on - this is Offline inference, also known as batch inference. Examples: benchmark evaluation and synthetic data generation. In this case the inference server is often not needed and the inference is run directly in the same program that sends the query (client and server in one application).

The 2 main use cases are often optimized for different performance metrics - the online inference use case requires a very low TTFT and low latency, whereas the offline inference requires high throughput. The combined prefill and decode token processing throughput is the key metric for any type of inference because it defines the total cost of the inference service. In the case of online inference, the better the combined throughput the more users can be served with the same hardware. For offline inference, it's clear that the faster the inference is done, the smaller the compute costs will be.

Batching

Processing the decoding stage one token at a time is extremely accelerator-inefficient. Batching multiple queries together improved the accelerator utilization and enables processing multiple requests at once.

The maximum possible batch size depends on how much memory is left after loading the model weights and filling the KV-cache with intermediate states.

Static batching

This is the naive straightforward batching where the first N queries are batched together - the problem here is that if many queries have finished generating they will have to wait for the longest to generate query to complete before they can be returned to the caller - greatly increasing the latency.

Continuous Batching or In-flight batching

Continuous Batching or In-flight batching is a process where the generation engine removes completed results as soon as they are done and replacing them with new queries, without waiting for the whole batch to complete. So that a sequence in position 0 in the batch could be generating its 10th token, while a sequence in position 1 in the batch could be just starting its first token generation, and position 3 is producing its last token.

This improves the response time, since a sequence that already finished can be returned immediately and there is no need for a new prompt to wait for the next batch to become available. Of course, if all of the compute is fully busy, and there are no new openings in the batch, then some requests will have to wait before the compute will start processing those.

Paged Attention

Paged Attention is very popular with inference servers as it allows for a very efficient accelerator memory utilization, by the virtue of approaching the accelerator memory like the OS memory using paging, which allowed dynamic memory allocation and prevents memory fragmentation.

Decoding methods

The main decoding methods are: Greedy decoding, Beam search and Sampling.

Greedy decoding

Greedy decoding is when the model always chooses the token with the highest probability. This is the fastest decoding

method but it doesn't always generate the best outcome, since it may choose a less ideal token path and miss out on a great future sequence of tokens.

One of the main issues with this method is creation of loops, where the same sentence is repeated again and again.

Beam search

Beam search overcomes the limitation of greedy decoding by generating multiple outputs at the same time, so instead of following the highest probability - with beam size of 3 it follows top 3 probabilities at each new token, then discards all but the 3 sub-paths out of 9 (`3*3`), that lead to the highest total probability of all tokens in the chain. Then at the end the path with the highest probability of all tokens is chosen.

This method is slower than greedy decoding because it has to generate n times more tokens and it requires n times more memory.

Sampling

Sampling-based decoding introduces randomness.

But, of course, choosing random words will not lead to a good result, so we still want greedy decoding like certainty but making it more interesting/alive by adding controlled randomness to it.

The most common sampling methods are:

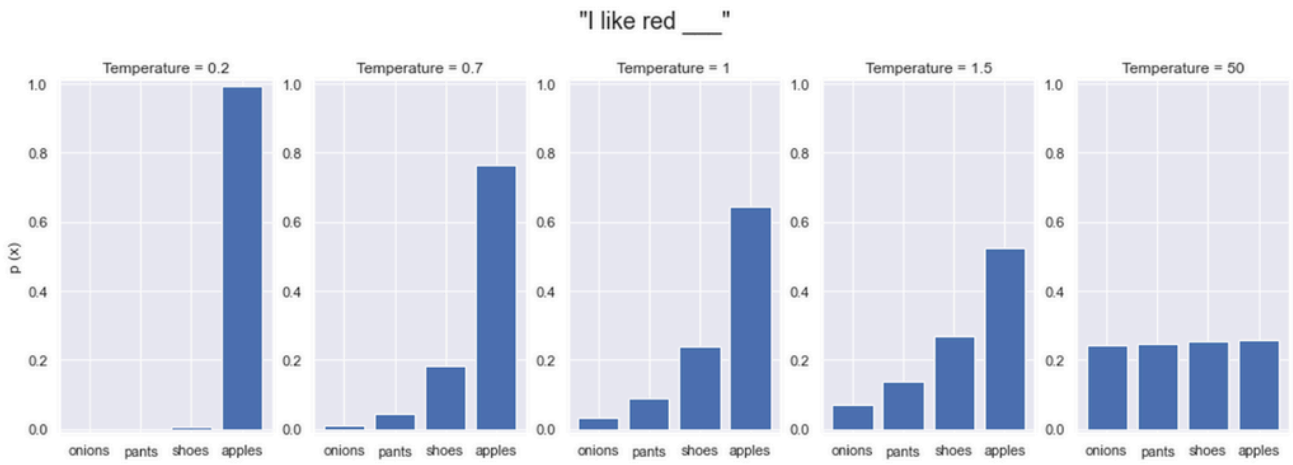
- **Top-K Sampling** method chooses the top k tokens based on their logit probability and then randomly picks one of those tokens.
- **Top-p Sampling** (also known as **nucleus sampling**) is like Top-K Sampling, but the K varies for each next token and is calculated by adding up the top token probabilities till the threshold `p` is reached. So if there are predictions that the model is much more certain about only those will be considered.

Temperature

Temperature is an independent logits transformation that can be combined with Top-K or Top-p sampling, both of them, or neither. It has the following impact depending on its value:

- `t==0.0`: literal softmax temperature division by zero is undefined. Some APIs use zero as shorthand for greedy decoding while others reject it, so use the API's explicit greedy mode when available.
- `0.0<t<1.0`: the probabilities are pushed further apart, so the closer to 0.0 the less randomness - somewhere between precise and balanced use cases.
- `t==1.0`: has no impact on sampling - the original training distribution is preserved here - balanced relevance and diversity use cases.
- `t>1.0`: the probabilities are pushed closer together, creating a lot more randomness - creative use cases.

The following set of plots should make this crystal clear:



(source)

To really understand the impact, the temperature factor typically gets applied to the log probabilities just before or as part of the Softmax operation.

```
scaled_logits = logits / temperature
probs = softmax(scaled_logits)
```

The softmax operation turns logit differences into probability ratios - when we divide by t<1.0, we make these differences larger, causing more extreme probability ratios and a more peaked distribution. When we divide by t>1.0, we make these differences smaller, causing more similar probability ratios and a more uniform distribution. As t approaches 0 from above, the highest logit dominates the distribution, but literal division by zero is avoided in practice.

For any positive temperature, scaling doesn't change the logit rank order, so a pure greedy argmax chooses the same token. It still changes sampling probabilities. With Top-K, the same K candidates remain eligible but their relative sampling probabilities change. With Top-p, temperature can change both which tokens enter the nucleus and their relative probabilities. Beam-search behavior depends on the implementation and its sequence scoring, so it shouldn't be assumed to be invariant. See the Transformers `TemperatureLogitsWarper` documentation.

There are no universal temperature values to copy from and you will have to experiment with each use case to find the values that work the best for your needs - though you surely will find people offering good baselines for different use cases if you search the Internet.

For more on decoding methods, see this HuggingFace blog.

Guided Text Generation

Also known as Structured Text Generation.

If the model can return its generated output in a specific format, rather than unrestricted format, you don't want the model to hallucinate invalid formats. For example, if you want a model to return a JSON dict, it should do just that.

The way this is accomplished is by using guided text generation. Instead of choosing a generated token with highest probability, the technique uses the next best token with highest probability that fits the next expected token sub-set. To elucidate with an example: if you want the model to generate a JSON list of strings like `["apples", "oranges"]` thus we expect:

```
["string", "string", ..., "string"]
```

```
123...
```

The first generated token has to be `[`. If the model got `"`, for example, instead of `[`, as the highest probability, and `[` had a lower probability - we want to pick the one with lower probability so that it'll be `[`.

Then the next generated token has to be `"`. If it's not, search for tokens with lower probabilities until `"` is found and choose that.

The third token has to be a valid string (i.e. not `[` or `"`).

And so on.

Basically, for each next token we need to know a subset of tokens that is allowed and choose one with the highest probability from that subset.

This is a very cool technique. Instead of trying to repair the generated output which is not always possible to match the expected format, we get the model to generate the correct output in the first place.

This technique has several costs:

- it slows down the generation - the more complex the schema it has to adhere to the slower it'll be at generating tokens. From measuring generation speed I found some structured text generation libraries perform much faster than others.
- it may contribute to model hallucination.

There are multiple implementations of this technique. As of 2026-08 vLLM supports four of them as structured output backends, and picks one automatically unless you override it:

- https://github.com/mlc-ai/xgrammar
- https://github.com/guidance-ai/llguidance
- https://github.com/dottxt-ai/outlines
- https://github.com/noamgat/lm-format-enforcer

You ideally want the implementations that have already been integrated into inference frameworks like vLLM and others.

Faster inference with guided generation

It's possible to use the schema to speed up inference as well. For example, consider this simple "profile" schema:

```
{
  "type": "object",
  "properties": {
    "name": { "type": "string"},
    "age": { "type": "integer"}
  },
  "required": ["name", "age"]
}
```

Since the schema has specific keys `name` and `age`, as soon as the model has predicted: `{"n` or `{"a` it doesn't need to perform an auto-regressive generation to come up with ``{"name": and{"age": `because both of these must lead to a specific unambiguous single outcome - so here it can perform a prefill instead of decoding and save a few slow steps as it knows 100% the next few tokens will beame": orge":` correspondingly. Clearly, this approach would be most beneficial when the schema has a lot of pre-determined keys and short generated values.

Speculative decoding

Also known as Speculative inference or Assisted generation.

Because it's very slow to generate tokens one a time, sometimes it is possible to cheat and speed things up by using a much smaller and faster draft model. So for example, your normal inference uses Llama-70B which would be quite slow, but we could use Llama-7b as a draft model and then we could verify the prediction is correct by doing it at once for all tokens.

Example: let's take a prompt `I'm turnin', turnin', turnin', turnin', turnin' around and all that I can see is just` and now:

1. use Llama-7b to predict `another lemon tree` auto-regressively, in 3 steps, but much faster than Llama-70b.
2. now use Llama-70b to run a batch of 3 prompts:

```
[...I can see is just]
[...I can see is just another]
[...I can see is just another lemon]
```

I shortened the full prompt for the sake of the demo with `...` - it should be there for real. And I'm pretending that each token is a full word here.

And now in a single step Llama-70B generates:

```
[...I can see is just] another
[...I can see is just another] lemon
[...I can see is just another lemon] tree
```

Now there could be multiple outcomes:

- if everything matches - in 3 short and 1 long step we generated the final result, instead of using 3 long steps.
- if only `another lemon` matched - we might still better off if it saved time.
- if nothing or little matched we wasted a bit of time.

Obviously, if instead of 3 tokens we had more tokens the savings are likely to be bigger.

Also, don't miss the fact that we did the same amount of compute here and then some, as compared to doing this generation with the large model normally, but the latency of this approach can be much better - so the user on average should get a better response time from your application using it - if the draft model is much smaller and still produces good predictions.

When there is a partial mismatch we can go back to the draft model and feed it all the matched tokens before the first mismatched token and the next good token predicted by the big model and get it to make a new fast prediction for the mismatching tail.

The draft model ideally should be trained on the same data (or least data from a similar distribution) and its tokenizer has to be the same as the large model.

Speculative decoding gives the highest return on input-grounded tasks, such as translation, summarization, document QA, multi-turn chat because in those tasks the range of possible outputs is much smaller and the draft model is much more likely to match the big model.

For the same reason it works best when used with greedy decoding, as there is the least amount of possible variations during generation. If not using greedy decoding, you will want to have the value of temperature close to 0.

Here is a good indepth dive into this subject: Assisted Generation: a new direction toward low-latency text generation.

One other much simpler solution for input-grounded tasks, is to use ngram prompt lookup decoding. In this approach there is no need for a draft model, instead the prompt is searched for matching strings to generate candidates. In some situations it's said to speed decoding up by 2x+.

Privacy-preserving inference

Most companies serving inference will run into a user privacy need. It should be safe for a user to submit a query w/o someone snooping on it. One solution would be an on-premise solution where the client runs the server themselves and then there is no privacy issue, but that most likely is going to expose provider's IP - model's weights and possibly code/algorithms. Therefore, there is a need for a fully encrypted generation - that is the computations are to be performed on client-encrypted data.

The solutions that address this need are called Privacy-Preserving Machine Learning (PPML).

One of the solutions is called Fully Homomorphic Encryption (FHE).

Have a look at one such implementation, concrete-ml that rewrites the model to be able to have the client run part of the model themselves, then the intermediary encrypted activations are sent to the server to perform the attention and then sent back to the client. Thus the provider retains part of their IP - and I suppose this part of IP prevents the client from stealing the full IP, since partial weights aren't enough to reconstruct the full model. This article goes into more details.

There are various other approaches, e.g. this paper: LLMs Can Understand Encrypted Prompt: Towards Privacy-Computing Friendly Transformers goes into a custom solution based on Secure Multi-Party Computation (MPC) and FHE and has a good reference list.

The problem with current solutions is the huge computational overhead - which greatly impacts the cost and latency. In the future ASIC solutions should address these issues.

Model parallelism

When a model can't fit onto a single accelerator or when it's more efficient to split the model across multiple accelerators even if it does fit but barely, the same Model Parallelism techniques from training apply to inference.

Tensor parallelism

Most of the time you are most likely to only run into Tensor Parallelism where the model weights are sharded across 2 to 8 accelerators. Ideally you want to try to fit the model into a single accelerator, because then it has the least amount of overhead during generation. But surprisingly you are likely to end up with higher decoding throughput if you use tensor parallelism - this is because it enables you to fit much larger batches and also because the `forward` call may be faster despite the additional comms between the accelerators. Of course, you will be getting this speed up at a cost of using more accelerators in some cases. So it's best to experiment, there will be use-cases where a higher tensor parallelism degree will give a better total throughput considering the same number of accelerators.

footnote: in my experiments TP=1 leads to the highest TTFT and lowest decoding throughput, as compared to TP>1. So if you're being requested to make the TTFT faster and the model fits, use smaller TP or TP=1. If you're being requested to make the decoding throughput faster, throw more accelerators at it with a higher TP degree.

Pipeline parallelism

Further, while tensor parallelism helps to lower latency, using Pipeline Parallelism could help increase the throughput. This is especially so for very large models where many accelerators have to be used anyway to even load the model's weights. If say you're using Llama 405B and TP=8 is used, then each accelerator has to all-reduce to 7 other accelerators, whereas with PP=8 each accelerator needs to communicate only with 2 other accelerators (`recv` the input from the previous stage and `send` the current output to the next stage), creating a much lower pressure on the networking layer and this can speed things up dramatically if the hardware supports it.

It's important to clarify here that PP can be superior to TP only if you use the full PP and not the naive PP. In the naive PP only one PP stage works at any given time so it'd perform worse than TP. To benefit from PP the inference framework needs to feeds all PP stages in parallel to perform full PP.

The other important thing about PP inference is that unlike training, there is no `backward` pass, thus there is no need to solve the inactivity bubble problem. There will be only a tiny overhead of filling the PP stages in the first few micro-batches.

And as with training you may find that some mix of TP and PP will lead to the best outcome (e.g. TP=4 + PP=4 for Llama 405B). So make sure to experiment and measure different configurations and pick the one that meets your needs.

Grounding

It's the process of giving the pre-trained model additional information that wasn't available during its training. For example input-grounded tasks give the model a lot of additional information in the prompt. Non zero-shot prompts ground the model in examples altering the default model behavior. Prompt-engineering is all about grounding the model to behave in a certain way during inference.

Retrieval Augmented Generation (RAG) is one of the main techniques for grounding models as it supplies the inference process with additional data that is relevant to the prompt. And the intention is that the model will give more significance to that information than the massive compressed information it was trained on.

Fine-tuning to a different knowledge domain is another grounding approach, we update the model to be grounded in a new dataset that could be quite distinct from the original domain of data the foundational model has been trained on.

Grounding can be thought of as providing context. As anybody can attest it's easier to answer a question when one understands the context of the question. The same applies with model generation. The better the context, the more relevant the generated output is.

In a multi-modal use case an image or a video supplied with the text prompt can be that grounding or a context.

Tasks

Input-grounded tasks

Input-grounded tasks are those where the generated response is derived mainly from the prompt, i.e. the main source of knowledge is contained in the prompt. These include:

- Translation
- Summarization
- Document QA
- Multi-turn chat
- Code editing
- Speech recognition (audio transcription)

Key inference performance metrics

There are two ways to look at performance metrics, the usual system metrics of latency and throughput, and the user-experience metrics: Time To First Token (TTFT) and Time Per Output Token (TPOT). Let's look at both pairs.

System performance metrics

Latency

Latency is the time it took to receive the complete response since a request was sent.

This includes the time to:

1. receive the request
2. pre-process the prompt (the prefill stage)
3. generate the new tokens of the response (the decoding stage)
4. send the response back to the client.

The time to receive the request and send the response is mostly the same with a small variation due to the differences in the length of the prompt and the generated response. These length variations should have a negligible impact to the total time.

The prefill stage processes prompt tokens in parallel, but longer prompts require more computation and KV-cache memory and therefore generally increase TTFT and reduce throughput. The effect depends on the model, batching, hardware, and server load.

The decoding stage is the one most impacted by the length of the generated response since each new token is generated as a separate step. Here the longer the response the longer the decoding stage will be.

If the server doesn't have enough capacity to process all current requests at once and has to queue some of them, then the wait time in the queue extends the latency by that time.

footnote: if you think of car traffic on the road, latency is the time it takes one to drive from point A to point B (e.g. home to office), including the speed limitations due to traffic lights, jams and legal limits.

Throughput

Throughput measures the ability of an inference server to process many requests in parallel and batch requests efficiently.

The definition of throughput could be defined by how many requests can be served concurrently, but since some requests get served much faster than others, so that several short requests could be served during a single long request, it makes sense to count the total rate of tokens generated across the system.

Thus a more common definition of **inference throughput is total tokens generated per second across the whole system.**

footnote: if you think of car traffic on the road, throughput is how many cars can move through a given road at any given time. The more lanes the road has and the higher the speed limit the higher the throughput of that road. But clearly some vehicles are short and some are long, so some sort of normalization is needed. For example, ferries calculate how many meters or feet of vehicles they can fit it and thus long vehicles pay more than short ones.

User experience metrics

While there are many characteristics an inference server can be judged by - like power usage, efficiency and cost, one could say that since the systems interface humans - the most important characteristics are all in the domain on having a smooth user experience. If the user experience is slow and choppy, the user will go to a competitor. Therefore the key needs are:

Time To First Token

Time To First Token (TTFT) is defined as the time that passed since the user hit the Submit button (or Enter) and the moment they have received a first word or a part of the word in return.

A very low Time To First Token (TTFT) is wanted. These days users are conditioned to expect from any application to start responding ideally faster than 1 second. Therefore the shorter the time the user has to wait before they start receiving the fist tokens the better. This becomes even more important for chatbots which are expected to be interactive. The length of TTFT is impacted by many elements, the key ones being the computation of the prefill stage (pre-processing the prompt) and whether the request got its processing immediately upon user request received or whether it had to wait in the queue.

It's important to observe that TTFT w/o a load on a server can be very different from when a server is under a heavy load. If normally the server sends the first token in 1 sec, if the server is already busy processing all the requests it can handle at once and there is a queue, the effective TTFT other than for the first few requests, could easily be much much longer. So usually one should measure an average TTFT and report it together with the number of concurrent requests sent during

the benchmark.

This is a non-trivial metric since depending on the prompt size the time will vary, so ideally you'd want to normalize it to the number of tokens in the prompt.

Time Per Output Token

Time Per Output Token (TPOT) is a per user metric. It measures how long does it take for a new token to be generated for a given user.

A relatively low Time Per Output Token (TPOT) is desired, but it doesn't have to be too low. This time ideally should be close to the reading speed of the human who sent the request. So for example if you serve first graders the TPOT can be relatively high, but the more educated the person is the lower the TPOT should be to achieve a smooth reading experience.

According to wiki there are 3 types of reading and the reading speed is measured in words per minute (WPM).

The average tokens per word varies with the language, somewhat with the tokenizer, and most of all with the kind of text being generated. Here let's consider an English tokenizer with about 1.5 tokens per word. Now we can convert words per minute (WPM) to tokens per minute (TPM).

And now we just need to divide by 60 to get Tokens Per Second (TPS) and invert to get time per output token (TPOT)

So `TPOT = 60 / (WPM*1.5)` in seconds

Reader	WPM	TPM	TPS	TPOT
Subvocal	250	375	6.25	0.16
Auditory	450	675	11.25	0.089
Visual	700	1050	18.75	0.057

Remember to change the 1.5 co-efficient to the actual word to tokens average ratio of your tokenizer, which you can measure on your own text:

```
import re, tiktoken
enc  = tiktoken.get_encoding("cl100k_base")
text = open("sample.txt").read()
print("tokens per word: ", len(enc.encode(text)) / len(re.findall(r"\S+", text)))
```

Measured on a paragraph of plain English this gives about 1.2 tokens per word on OpenAI's `cl100k` and `o200k` tokenizers, and the same on GPT-2's older 50k vocab - for prose the choice of tokenizer barely moves it. What does move it is the kind of text the model emits: the same measurement on Python code gives about 3.3 tokens per word. So if you serve code completions rather than prose, a 1.5 co-efficient sets your TPOT target more than 2x too lax - you'd budget 0.16 secs per token for a 250 WPM reader where the code case needs closer to 0.07.

As you can see TPOT is an awkward value to track and think of in one's head, so **once you know your targeted TPOT it's better to convert it to Tokens Per Seconds (TPS) and track that instead**.

Therefore in this example if your system can generate a sustainable 20 tokens per second per request your clients will be satisfied since that system will be able to keep up even with the super-fast readers at 700 words per minute.

And there, of course, will be users who would prefer to wait till the generation is complete before they would start reading the response. In which case faster is better.

Depending on the type of generation, the following is likely to apply:

1. Image - all-at-once
2. Text - as fast as user's reading speed or all-at-once if they prefer not to have moving parts before they start reading
3. Audio - as fast as user's listening speed
4. Video - as fast as user's watching speed

If this is an offline system that doesn't interface individual humans and there are just batches of requests processed these metrics make no difference, but latency and throughput are the key ones.

Simplified performance metrics

As you can tell the metrics discussed so far have a lot of overlap in them. Practically we can reduce all of them to just these 2 metrics: Prefill throughput and Decode throughput - and probably how many parallel requests per second the system can handle.

Prefill throughput

This is how fast the system can pre-process the prompt - in tokens per second.

Assuming there is a negligible overhead of receiving and sending the request, in the absence of a queue where the incoming request gets immediately workedon TTFT is really the number of tokens in the prompt divided by the prefill tokens per seconds plus the time to generate the first token (which we can ignore as it'd be very fast).

If there is a queue then prefill throughput isn't enough, because then TTFT can be much longer as one has to add the time the request spent in the queue.

Decode throughput

This is how fast the system generates response tokens - in tokens per second.

This addresses, both the throughput and Time Per Output Token metrics.

The response latency then is the number of tokens in the prompt divided by the prefill throughput plus the number of generated tokens divided by the decode throughput.

More metric notes

Accelerator utilization

Accelerator utilization - either percentage or power measurement is a good indicator of whether your setup uses the accelerators efficiently. For example, if you use NVIDIA GPUs and you `watch -n 0.5 nvidia-smi` and you see you're at 10% "gpu util" while massively bombarding the inference server with many requests that usually means that either the inference server is very inefficient (e.g. spends a lot of time copying things back-n-forth) or it could be that the clients are inefficient at how they receive the data (i.e. too much IO blocking).

footnote: when I first wrote a simple benchmark using the openai client it worked fine at a low concurrency, but at a higher concurrency the inference server dropped its gpu util to 6-7%. After I replaced the client with `aiohttp` API it went up to 75%. Therefore beware that it's your benchmark that could be the culprit of bad performance reports and not the server.

This is somewhat of an equivalent of using TFLOPS to measure training efficiency.

In the ideal case you want your accelerator utilization to be as high as possible. Beware that at least for NVIDIA GPUs `gpu util` isn't what you might think it is, but if it reports a very low percentage it's good enough of a signal to know that there is definitely an inefficiency problem.

Percentiles

If you read benchmarks and run into p50, p75, p90, p95, and p99, these percentiles are values in an ordered distribution.

A p95 value is the threshold at or below which 95% of observations fall; the remaining 5% are above it. The desirable tail depends on the metric. For latency, lower is better, so p95 describes high-tail latency. For throughput, higher is better, so p5 is often the useful lower-tail guarantee: about 95% of observations are at or above p5. See the Grafana k6 percentile documentation.

For example, let's look at partial output of a system loading report generated by k6 on an inference server:

```
http_req_duration..: avg=13.36s   min=12.54s  med=13.31s   max=14.12s   p(90)=13.79s
p(95)=13.83s
http_req_receiving.: avg=27.98µs  min=15.16µs med=21.6µs   max=98.13µs  p(90)=44.98µs
p(95)=59.2µs
http_req_sending...: avg=133.8µs  min=20.47µs med=75.39µs  max=598.04µs p(90)=327.73µs
p(95)=449.65µs
```

In the first line, 50% of responses completed in at most 13.31 seconds, 90% in at most 13.79 seconds, and 95% in at most 13.83 seconds. The slowest 5% took more than 13.83 seconds, up to the observed maximum of 14.12 seconds.

The same interpretation applies to the other lines. For any metric, a correctly ordered sample must satisfy `min <= p50 <= p90 <= p95 <= max`; whether a higher value is better or worse depends on what is measured.

Percentiles summarize tails without letting one extreme observation dominate the report. They do not make the omitted tail unimportant: a p95 latency still means that 5% of requests were slower, which can represent many users at production scale.

Please refer to Percentile for a much more indepth explanation.

Speeding up model loading time

When serving in production it might be OK to let the model takes its loading time since it happens once and then the server runs for days, so this overhead is amortized over many days. But when doing research, development and testing it's critical that the inference server starts serving really fast.

Sometimes the overhead is just loading to CPU and then moving the tensors to the accelerators, at other times there is an additional need to shard the tensors for multiple accelerators to perform TP and PP.

Various approaches are used for that - most involve some sort of pre-sharding and caching, with a subsequent direct loading onto GPU.

For example:

- vLLM supports the `--load-format` flag, where one could choose options like `npcache` (numpy format caching) or `tensorizer` using CoreWeave's Tensorizer. (recipe and, of course, if you use TP>1 you want to pre-shard the weights once.
- TensorRT-LLM requires the user to build a model engine for each specific use-case and loads the pre-made shards at run time (unless you're using the simplified API which will build the model engine on the fly on every server start).

Benchmarks

You can write your own benchmark as explained in key inference performance metrics or use an existing one.

At the moment I mainly use TTFT and decode throughput benchmarks. TTFT measures the client-observed time from sending the request until receiving the first non-empty generated chunk. The inter-token decode rate covers the interval between receiving the first and last generated tokens. Here is the relevant snippet using the `openai` client completions

API:

```
# ... create client, prompt, and model ...
request_started_time = time.perf_counter()
decode_text = ""
first_token_time = None
last_token_time = None
completion = client.completions.create(model=model, prompt=prompt, stream=True)
for chunk in completion:
    if chunk.choices:
        text = chunk.choices[0].text
        if text:
            now = time.perf_counter()
            if first_token_time is None:
                first_token_time = now
            last_token_time = now
            decode_text += text

if first_token_time is None:
    raise RuntimeError("The response contained no output tokens")

ttft = first_token_time - request_started_time
output_tokens = len(tokenizer.encode(decode_text, add_special_tokens=False))
inter_token_count = max(output_tokens - 1, 0)
inter_token_time = last_token_time - first_token_time
decode_throughput = (
    inter_token_count / inter_token_time
    if inter_token_count > 0 and inter_token_time > 0
    else None
)
```

The subtraction in `output_tokens - 1` matches the measured interval, which starts after the first token has arrived. This client-side estimate assumes streamed chunks correspond closely enough to token arrivals; an API that batches multiple tokens into a chunk needs token-level or server-side timestamps for precise inter-token latency.

TTFT is not pure prefill time. It includes request transport, queueing, scheduling, prompt processing, first-token generation, and response transport. Dividing the prompt-token count by TTFT therefore does not produce prefill throughput; that metric requires server-side instrumentation that isolates prompt processing. See NVIDIA's LLM inference metric definitions and MLPerf Inference's TTFT and TPOT definitions.

Of course, like any serious benchmark, you want to run this multiple times to get realistic numbers, as the variance between single runs can be quite large.

note: I've discovered that when I use the openAI client it doesn't scale well and with many concurrent requests the openAI client creates a bottleneck and doesn't measure the real server performance - I am yet to figure out if it's an issue in my code or the openAI client or how it interacts with vLLM server - I'm investigating here https://github.com/vllm-project/vllm/issues/7935 - I found that this version of the client, rewritten to use `aiohttp` scales really well - so I switched to using it.

Here are some good starting points for load testing:

- https://github.com/vllm-project/vllm/blob/main/benchmarks/benchmark_throughput.py - my favorite tool so far

- https://github.com/grafana/k6 - useful for load testing to simulate multiple concurrent clients - uses JavaScript clients.

What I'm missing right now is a tool to measure the highest concurrency the server can handle.

Inference frameworks

There are many dozens of inference frameworks and more emerging every week, so it'd be very difficult to list them all. So this here you will find a starter list of a handful of inference frameworks that might be a good fit for your needs, but do check out other frameworks if the ones listed here don't satisfy your needs.

This section is trying hard to be neutral and not recommend any particular frameworks, since even if I was able to try them all out, there is no way I could possible guess which framework will work best for which user/company.

vLLM

vLLM

DeepSpeed-FastGen

DeepSpeed-FastGen from the DeepSpeed team.

TensorRT-LLM

TensorRT-LLM (also integrated what used to be `FasterTransformer`)

Supports only NVIDIA gpus.

SGLang

SGLang

llama.cpp

llama.cpp - local / single-box serving (CPU, CUDA including datacenter GPUs, Metal, ...); not a multi-tenant production peer to vLLM/SGLang.

LightLLM

LightLLM

LMDeploy

LMDeploy

MLC-LLM

MLC-LLM

If your favorite inference framework isn't listed please make a PR and add it.

Accelerator-specific frameworks

Most inference frameworks obviously support NVIDIA CUDA. Many also support AMD ROCm and Intel Gaudi - increasingly via vendor plugins to the mainstream frameworks (e.g. vLLM's hardware-plugin architecture) rather than as separate stacks.

The main accelerator-specific software stacks are:

- **Intel Gaudi (HPU)**: the vllm-gaudi plugin (the older HabanaAI/vllm-fork is being retired), plus Optimum for Intel Gaudi for the HF `transformers`/`diffusers` ecosystem.
- **AWS Trainium/Inferentia**: the AWS Neuron SDK, which serves through standard vLLM APIs and Optimum Neuron.
- **Google TPU**: tpu-inference - the vLLM TPU plugin unifying JAX and PyTorch, and the successor to the now-archived JetStream; MaxText is the JAX-native alternative.
- **Modular MAX**: MAX - a portable serving stack that also runs on CPUs and NVIDIA/AMD GPUs.
- **Apple Silicon**: MLX / mlx-lm, plus llama.cpp's Metal backend.
- **Tenstorrent**: tt-metal and its vLLM integration.

Because accelerator support increasingly ships as a plugin to a mainstream framework, always check whether your framework of choice already supports your accelerator before reaching for a vendor-specific stack.

Finally, some vendors expose inference only as a hosted service on their own hardware, rather than as an installable framework - e.g. Cerebras Inference, Groq and SambaNova.

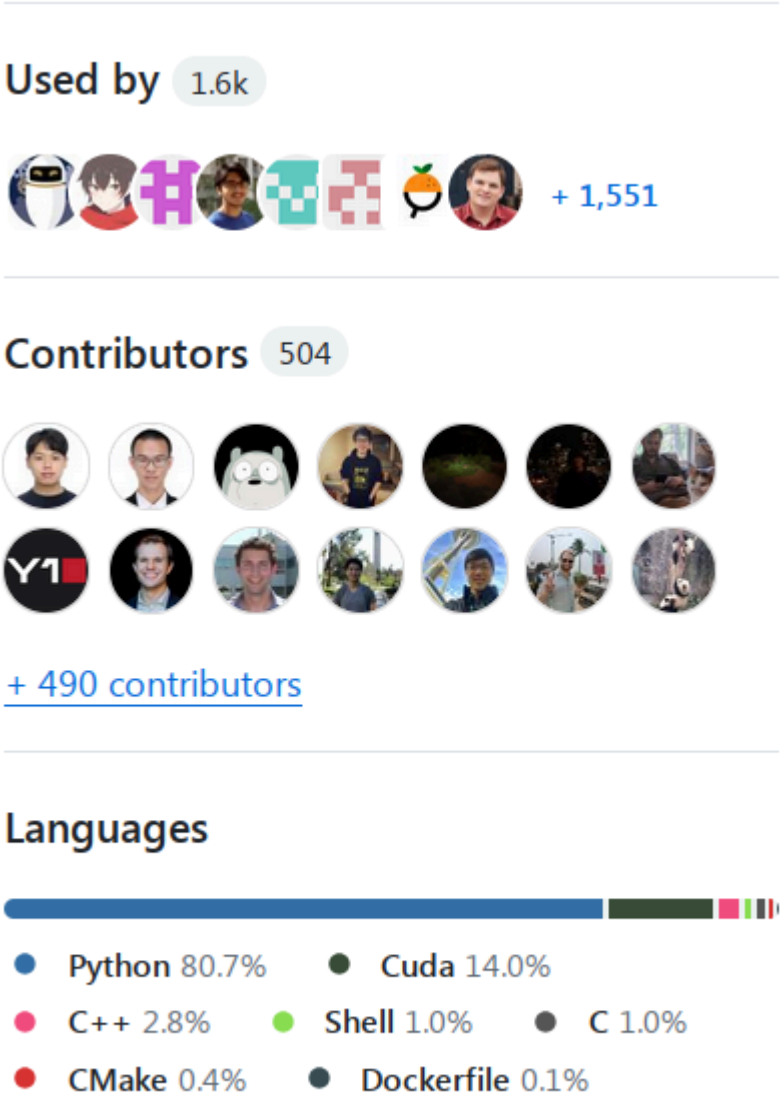
How to choose an inference framework

To choose the most suitable inference framework you need to answer at least the following questions:

1. Does the framework have the features that you need? Be careful here, some frameworks list that they support feature A, but when you try to use it, it's not well integrated or works really slowly.
2. Does the framework have a permissive license that meets your current and future needs? In practice we have seen that frameworks with licenses that go against commercial use are likely to be rejected by the community. For example, HF's TGI tried to charge for commercial use and it backfired - their licenses got reverted under community pressure, but TGI has never recovered and the development has stopped.
3. Does the framework have a thriving community of contributors? Go to the framework's github repo and check how many contributors it has - if it's very few I'd be concerned as thriving frameworks usually tend to invite contributions and that means that even if the core contributors don't have the time some feature, some contributors might do it for you.
4. Does the framework have a high adoption? github stars are often a good indication, but sometimes it can be hyped up via smart marketing moves. So seek out other signals - e.g. `Used by` count on the framework's repo's main page on github - these are real numbers. Lots of PRs and Issues is another flag. Then search the web for how many articles are written about the given framework.
5. Are the framework maintainers responsive to Issues and PRs? Some frameworks will ignore many Issues and even PRs. Check the count of how many PRs and Issues not being addressed. A high outstanding open Issues is a difficult signal - from one side it means this is a popular project, from the other side it means the developer team and contributors can't cope with the needs of its users.
6. While the majority of ML inference frameworks are written in Python, with some sprinkling of C++ or Triton for fused kernels, some aren't written in Python. (e.g. NVIDIA's TensorRT-LLM is 99% C++, llama.cpp is C++/CUDA). If something doesn't work the way you need it to and you filed an Issue and it's not being addressed, will you be able to get your hands dirty and modify the framework to do what you need?
7. The other issue you may run into is that some frameworks don't want your PRs where you implemented missing features or made improvements and then you will end up maintaining a fork, which can be extremely difficult if you want to continue syncing with the upstream and cause a lot of pain to your developers.
8. Run some sort of load benchmarks for the desired workloads to know if the performance is adequate.
9. Will you want to choose the best cost-effective accelerator down the road or are you OK being locked in into a specific vendor? For example, a framework from NVIDIA isn't likely to support any other accelerators besides NVIDIA's. Same goes for AMD and Intel.

For example, here is a snapshot of vLLM's stats as of 2024-08-24, which is one of the most popular inference frameworks as

of 2026-08.



You can see that it is used by many github repositories, it has a lot of contributors and that it's written mainly in Python. So it should be very easy to find this information about any inference framework you may consider. This was just an example and not an endorsement of vLLM.

Inference Chips

Besides general purpose accelerators some vendors have been working special ASICs that are designed to do Inference-only.

Groq

- Groq

Resources

- A Survey on Efficient Inference for Large Language Models (2024)

Debugging and Troubleshooting

Guides

- Debugging PyTorch programs
- Diagnosing Hangings and Deadlocks in Multi-Node Multi-GPU Python Programs
- Network Debug
- Troubleshooting NVIDIA GPUs
- Underflow and Overflow Detection

Tools

- Debug Tools
- torch-distributed-gpu-test.py - this a `torch.distributed` diagnostics script that checks that all GPUs in the cluster (one or many nodes) can talk to each other and allocate gpu memory.
- NicerTrace - this is an improved `trace` python module with multiple additional flags added to the constructor and more useful output.

Debugging PyTorch Programs

Fast debug of PyTorch models

Faster Debug and Development with Tiny Models, Tokenizers and Datasets

If you're debugging problems and develop with full sized models and tokenizers you're likely not working in a very efficient way. Not only it's much more difficult to solve problem, the amount of waiting to get the program to restart and to get to the desirable point can be huge - and cumulatively this can be a huge drain on one's motivation and productivity, not talking about the resolution taking much longer, if at all.

The solution is simple:

Unless you're testing the quality of a model, always use a tiny random model with potentially tiny tokenizer.

Moreover, large models often require massive resources, which are typically expensive and can also can make a debugging process super complicated. For example any debugger can handle a single process, but if your model doesn't fit and require some sort of parallelization that requires multiple processes - most debuggers will either break or have issue giving you what you need. The ideal development environment is one process and a tiny model is guaranteed to fit on an even cheapest single smallest consumer GPU available. You could even use the free Google Colab to do development in a pinch if you have no GPUs around.

So the updated ML development mantra then becomes:

- the larger the model the better the final product generates
- the smaller the model the quicker the final product's training can be started

footnote: the recent research shows that larger isn't always better, but it's good enough to convey the importance of my communication.

Once your code is working, do switch to the real model to test the quality of your generation. But even in this case still try first the smallest model that produces a quality result. Only when you can see that the generation is mostly right use the largest model to validate if your work has been perfect.

Making a tiny model

Important: given their popularity and the well designed simple API I will be discussing HF `transformers` models. But the same principle can be applied to any other model.

TLDR: it's trivial to make a tiny HF `transformers` model:

1. Fetch the config object of a full size model
2. Shrink the hidden size and perhaps a few other parameters that contribute to the bulk of the model
3. Create a model from that shrunken config
4. Save this model. Done!

footnote: It's critical to remember that this will generate a random model, so don't expect any quality from its output.

footnote: These notes were written with HF Transformers models in mind. If you're using a different modeling library you may have to adapt some of these things.

Now let's go through the actual code and convert "google/mt5-small" into its tiny random counterpart.

```
from transformers import MT5Config, MT5ForConditionalGeneration
```

```
mname_from = "google/mt5-small"
mname_very_small = "mt5-tiny-random"

config = MT5Config.from_pretrained(mname_from)

config.update(dict(
    d_model=64,
    d_ff=256,
))
print("new config", config)

very_small_model = MT5ForConditionalGeneration(config)
print(f"num of params {very_small_model.num_parameters()}")

very_small_model.save_pretrained(mname_very_small)
```

As you can see it's trivial to do. And you can make it even smaller if you don't need the hidden size to be at least 64. For example try 8 - you just need to make sure that the number of attention heads isn't larger than hidden size.

Also please note that you don't need any GPUs to do that and you could do this even on a huge 176B parameter model like BLOOM-176B. Since you never load the actual original model, except its config object.

Before modifying the config you can dump the original parameters and choose to shrinks more dimensions. For example, using less layers makes it even smaller and easier to debug. So here is what you can do instead:

```
config.update(dict(
    d_model=64,
    d_ff=256,
    d_kv=8,
    num_layers=8,
    num_decoder_layers=8,
    num_heads=4,
    relative_attention_num_buckets=32,
))
```

Note that this encoder-decoder model takes both layer counts. `num_decoder_layers` falls back to `num_layers` only when it isn't set, and a config loaded from a real checkpoint already has it set, so trimming `num_layers` alone would leave the decoder at its full depth. Other encoder-decoder architectures split it differently - BART, Marian, Pegasus and Whisper name the pair `encoder_layers` and `decoder_layers` - but the rule is the same: shrink both.

The original "google/mt5-small" model file was 1.2GB. With the above changes (and vocab shrinking as explained in the following sections) we got it down to 126MB.

If you're dealing with a multi-level nested config, you will have to update each sub-level's config object separately. For example in IDEFICS we have 1 main and 2 nested objects:

```
config
config.perceiver_config
config.vision_config
```

If you wanted to shrink this model you'd want to update `config` and `config.vision_config` with smaller values:

```
config.update(dict(
    hidden_size=64,
    intermediate_size=37,
    num_hidden_layers=5,
    num_attention_heads=4,
    max_position_embeddings=64,
    max_sequence_length=64,

))
# sub object needs to be updated directly
config.vision_config.update(dict(embed_dim=64))
```

See idefics-make-tiny-model.py for a fully working script (I didn't bother adding the vocab shrinking as I'm just demonstrating how to update nested config objects here).

A related trap: architectures that interleave block types spell the pattern out in a `layer_types` list with one entry per layer. Qwen/Qwen-AgentWorld-35B-A3B carries `40` `linear_attention`/`full_attention` entries beside its `"num_hidden_layers": 40`, and the two have to be trimmed together - and, as in the nested case above, both keys live under `text_config`.

A mismatch fails in two different ways. Save the shrunken config and load it back, and it refuses outright:

```
StrictDataclassClassValidationError: Class validation error for validator 'validate_layer_type':
    ValueError: `num_hidden_layers` (2) must be equal to the number of `layer_types` (40)
```

Mutate the config object in memory only, and nothing is raised - the model quietly builds your 2 layers from the first 2 entries of the untouched list, and since this model's full attention block first appears at index 3, you get two linear-attention layers and none of the block you meant to keep. So trim the list in lockstep:

```
config.text_config.update(dict(
    num_hidden_layers=4,
    layer_types=config.text_config.layer_types[:4],
))
```

If you're editing `config.json` directly, you can instead delete the `layer_types` entry and let the config regenerate the pattern from `full_attention_interval` at the new depth. Either way, pick a depth that spans one full period of the pattern - 4 rather than 2 for this model - or the variation you were keeping disappears.

We can then further halve our tiny model size by converting the model to fp16 or bf16 (depending on the goal) before saving it:

```
very_small_model.half() # convert to fp16
#very_small_model.bfloat16() # convert to bf16
very_small_model.save_pretrained(mname_very_small)
```

this takes us to 64MB file.

So you could stop here and your program will start much much faster already.

And there is one more step you could do to make it truly tiny.

What we haven't shrunken so far is the vocabulary dimension so 64x250k (hidden*vocab) is still huge. Granted this 250k vocab model is not typical - normally models' vocab is ~30-50k, but even 30k is a lot if we want the model to be truly tiny.

So next we will look into various techniques to shrinking the tokenizer, as it defines our vocab size.

Making a tiny tokenizer

This task varies between a relatively simple procedure and a much more complex workout depending on the underlying tokenizer.

The following recipes have come from a few awesome tokenizer experts at HuggingFace, which I then adapted to my needs.

You probably don't really need to understand how these work until you actually need them, therefore if you're reading this for the first time you can safely jump over these to Making a tiny model with a tiny tokenizer.

Anthony Moi's version

Anthony Moi's tokenizer shrinker:

```
import json
from transformers import AutoTokenizer
from tokenizers import Tokenizer

vocab_keep_items = 5000
mname = "microsoft/deberta-base"

tokenizer = AutoTokenizer.from_pretrained(mname, use_fast=True)
assert tokenizer.is_fast, "This only works for fast tokenizers."
tokenizer_json = json.loads(tokenizer._tokenizer.to_str())
vocab = tokenizer_json["model"]["vocab"]
if tokenizer_json["model"]["type"] == "BPE":
    new_vocab = { token: i for token, i in vocab.items() if i < vocab_keep_items }
    merges = tokenizer_json["model"]["merges"]
    new_merges = []
    for i in range(len(merges)):
        a, b = merges[i].split()
        new_token = "".join((a, b))
        if a in new_vocab and b in new_vocab and new_token in new_vocab:
            new_merges.append(merges[i])
    tokenizer_json["model"]["merges"] = new_merges
elif tokenizer_json["model"]["type"] == "Unigram":
    new_vocab = vocab[:vocab_keep_items]
elif tokenizer_json["model"]["type"] == "WordPiece" or tokenizer_json["model"]["type"] ==
"WordLevel":
    new_vocab = { token: i for token, i in vocab.items() if i < vocab_keep_items }
else:
    raise ValueError(f"don't know how to handle {tokenizer_json['model']['type']}")
tokenizer_json["model"]["vocab"] = new_vocab
tokenizer._tokenizer = Tokenizer.from_str(json.dumps(tokenizer_json))
```

```
tokenizer.save_pretrained(".")
```

I later discovered that gpt2 seems to have a special token `"<|endoftext|>"` stashed at the very end of the vocab, so it gets dropped and code breaks. So I hacked it back in with:

```
if "gpt2" in mname:
        new_vocab = { token: i for token, i in vocab.items() if i < vocab_keep_items-1 }
        new_vocab["<|endoftext|>"] = vocab_keep_items-1
    else:
        new_vocab = { token: i for token, i in vocab.items() if i < vocab_keep_items }
```

Lysandre Debut's version

Lysandre Debut' shrinker using `train_new_from_iterator`:

```
from transformers import AutoTokenizer

mname = "microsoft/deberta-base" # or any checkpoint that has a fast tokenizer.
vocab_keep_items = 5000

tokenizer = AutoTokenizer.from_pretrained(mname)
assert tokenizer.is_fast, "This only works for fast tokenizers."
tokenizer.save_pretrained("big-tokenizer")
# Should be a generator of list of texts.
training_corpus = [
    ["This is the first sentence.", "This is the second one."],
    ["This sentence (contains #) over symbols and numbers 12 3.", "But not this one."],
]
new_tokenizer = tokenizer.train_new_from_iterator(training_corpus, vocab_size=vocab_keep_items)
new_tokenizer.save_pretrained("small-tokenizer")
```

but this one requires a training corpus, so I had an idea to cheat and train the new tokenizer on its own original vocab which gave me:

```
from transformers import AutoTokenizer

mname = "microsoft/deberta-base"
vocab_keep_items = 5000

tokenizer = AutoTokenizer.from_pretrained(mname)
assert tokenizer.is_fast, "This only works for fast tokenizers."
vocab = tokenizer.get_vocab()
training_corpus = [ vocab.keys() ] # Should be a generator of list of texts.
new_tokenizer = tokenizer.train_new_from_iterator(training_corpus, vocab_size=vocab_keep_items)
new_tokenizer.save_pretrained("small-tokenizer")
```

which is almost perfect, except it now doesn't have any information about the frequency for each word/char (that's how

most tokenizers compute their vocab, which if you need this info you can fix by having each key appearing `len(vocab) - ID times`, i.e.:

```
training_corpus = [ (k for i in range(vocab_len-v)) for k,v in vocab.items() ]
```

which will make the script much much longer to complete.

But for the needs of a tiny model (testing) the frequency doesn't matter at all.

Hack the tokenizer file approach

Some tokenizers can be just manually truncated at the file level, e.g. let's shrink Llama2's tokenizer to 3k items:

```
# Shrink the orig vocab to keep things small (just enough to tokenize any word, so
letters+symbols)
# ElectraTokenizerFast is fully defined by a tokenizer.json, which contains the vocab and the
ids,
# so we just need to truncate it wisely
import subprocess
import shlex
from transformers import LlamaTokenizerFast

mname = "meta-llama/Llama-2-7b-hf"
vocab_keep_items = 3000

tokenizer_fast = LlamaTokenizerFast.from_pretrained(mname)
tmp_dir = f"/tmp/{mname}"
tokenizer_fast.save_pretrained(tmp_dir)
# resize tokenizer.json (vocab.txt will be automatically resized on save_pretrained)
# perl  -0777 -pi -e 's|(2999).*|$1},"merges": []}}|msg' tokenizer.json # 0-indexed, so
vocab_keep_items-1!
closing_pat = '},"merges": []}}'
cmd = (f"perl -0777 -pi -e 's|({vocab_keep_items-1}).*|$1{closing_pat}|msg'
{tmp_dir}/tokenizer.json")
#print(f"Running:\n{cmd}")
result = subprocess.run(shlex.split(cmd), capture_output=True, text=True)
# reload with modified tokenizer
tokenizer_fast_tiny = LlamaTokenizerFast.from_pretrained(tmp_dir)
tokenizer_fast_tiny.save_pretrained(".")
```

Please remember that the outcome is only useful for functional testing - not quality work.

Here is the full version of make_tiny_model.py which includes both the model and the tokenizer shrinking.

SentencePiece vocab shrinking

First clone SentencePiece into a parent dir:

```
git clone https://github.com/google/sentencepiece
```

Now to the shrinking:

```
# workaround for fast tokenizer protobuf issue, and it's much faster too!
os.environ["PROTOCOL_BUFFERS_PYTHON_IMPLEMENTATION"] = "python"

from transformers import XLMRobertaTokenizerFast

mname = "xlm-roberta-base"

# Shrink the orig vocab to keep things small
vocab_keep_items = 5000
tmp_dir = f"/tmp/{mname}"
vocab_orig_path = f"{tmp_dir}/sentencepiece.bpe.model" # this name can be different
vocab_short_path = f"{tmp_dir}/spiece-short.model"
# HACK: need the sentencepiece source to get sentencepiece_model_pb2, as it doesn't get installed
sys.path.append("../sentencepiece/python/src/sentencepiece")
import sentencepiece_model_pb2 as model
tokenizer_orig = XLMRobertaTokenizerFast.from_pretrained(mname)
tokenizer_orig.save_pretrained(tmp_dir)
with open(vocab_orig_path, 'rb') as f: data = f.read()
# adapted from https://blog.ceshine.net/post/trim-down-sentencepiece-vocabulary/
m = model.ModelProto()
m.ParseFromString(data)
print(f"Shrinking vocab from original {len(m.pieces)} dict items")
for i in range(len(m.pieces) - vocab_keep_items): _ = m.pieces.pop()
print(f"new dict {len(m.pieces)}")
with open(vocab_short_path, 'wb') as f: f.write(m.SerializeToString())
m = None

tokenizer_fast_tiny = XLMRobertaTokenizerFast(vocab_file=vocab_short_path)
tokenizer_fast_tiny.save_pretrained(".")
```

Making a tiny model with a tiny tokenizer

So now you can shrink the vocab size to as small as the tokenizer allows, that is you need to have at least enough tokens to cover the target alphabet and special characters, and usually 3-5k tokens is more than enough. Sometimes you could make it even small, after all the original ASCII charset has only 128 characters.

If we continue the MT5 code from earlier in this chapter and add the tokenizer shrinking code from the previous section, we end up with this script mt5-make-tiny-model.py and when we run it - our end model file is truly tiny - 3.34MB in size! As you can see the script also has code to validate that the model can actually work with the modified tokenizer. The results will be garbage, but the intention is to test that the new model and the tokenizer are functional.

Here is another example fsmt-make-super-tiny-model.py - here you can see I'm creating a totally new tiny vocab from scratch.

I also recommend to always store the building scripts with the model, so that you could quickly fix things or make similar versions of the model.

Also be aware that since HF `transformers` needs tiny models for their testing, you are very likely to already find one for each architecture available mostly from https://huggingface.co/hf-internal-testing (except they didn't include the code of how they were made, but you can now figure it out based on these notes).

Another hint: if you need a slightly different tiny model, you can also start with an already existing tiny model and adapt it instead. Since it's random it's really only about getting the right dimensions. For example if the tiny model you found has 2 layers but you need 8, just resave it with this larger dimension and you're done.

Making a tiny dataset

Similar to models and tokenizers it helps to have a handy tiny version of a dataset you work with a lot. As usual this won't help with quality testing, but it's perfect for launching your program really fast.

footnote: the impact of using a tiny dataset won't be as massive as using a tiny model, if you're using already pre-indexed Arrow file datasets, since those are already extremely fast. But say you want the iterator to finish an epoch in 10 steps. Instead of editing your code to truncate the dataset, you could just use a tiny dataset instead.

This process of making a tiny dataset is somewhat more difficult to explain because it'd depend on the builder of the original dataset, which can be quite different from each other, but perhaps you can correlate my recipes to your datasets.

But the concept is still very simple:

1. Clone the full dataset git repo
2. Replace its full data tarball with a tiny one that contains just a few samples
3. Publish the tiny result as data files the Hub can load without a custom script (Parquet, Arrow, JSON/JSONL, CSV, or an image/text folder) plus a dataset card - Done!

footnote: as of `datasets 4.0`, Hub-hosted Python loading scripts are no longer executed by `load_dataset`. Keep any builder/unpacker `.py` in the repo as documentation of how the tiny set was made and for local regeneration, but what remote users load must be the data files themselves. See Dataset repository structure.

Here are some examples:

- stas/oscar-en-10k
- stas/c4-en-10k
- stas/openwebtext-10k

In all of these I took the original tarball, grabbed the first 10k records, tarred it back, used this smaller tarball and that was that. The rest of the builder script remained mostly the same.

And here are some examples of synthetic datasets, where instead of just shrinking the original tarball, I untar'ed it, manually chose the representative examples and then wrote a script to build any size of desired dataset based on those few representative samples:

- stas/general-pmd-synthetic-testing and the unpacker
- stas/cm4-synthetic-testing - and the unpacker

These are also the complex examples where each sample is more than a text entry, but may have multiple text entries and images as well.

The unpacker is what expands each complex multi-record sample into its own sub-directory, so that now you can easily go and tweak it to your liking. You can add image, remove them, make text records smaller, etc.. You will also notice that I'm shrinking the large images into tiny 32x32 images, so again I'm applying the important principle of tiny across all dimensions that don't break the requirements of the target codebase.

And then the main script uses that structure to build a dataset of any desired length.

And here is for example the instructions of deploying these scripts for stas/general-pmd-synthetic-testing:

```
# prep dataset repo
https://huggingface.co/new-dataset => stas/general-pmd-synthetic-testing
git clone https://huggingface.co/datasets/stas/general-pmd-synthetic-testing
```

```
cd general-pmd-synthetic-testing

# select a few seed records so there is some longer and shorter text, records with images and
without,
# a few variations of each type
rm -rf data
python general-pmd-ds-unpack.py --dataset_name_or_path \
general_pmd/image/localized_narratives__ADE20k/train/00000-00002 --ids 1-10 --target_path data

cd data

# shrink to 32x32 max, keeping ratio
mogrify -format jpg -resize 32x32\> */*jpg

# adjust one record to have no image and no text
cd 1
rm image.jpg text.txt
touch image.null text.null
cd -

cd ..

# create tarball
tar -cvzf data.tar.gz data

# complete the dataset repo
echo "This dataset is designed to be used in testing. It's derived from general-pmd/
localized_narratives__ADE20k \
dataset" >> README.md

# test dataset - it must load from plain data files, with no remote loading script
cd ..
python -c 'from datasets import load_dataset;
print(load_dataset("general-pmd-synthetic-testing"))'
```

I also recommend to always store the building scripts with the dataset, so that you could quickly fix things or make similar versions of the dataset.

Similar to tiny models, you will find many tiny datasets under https://huggingface.co/hf-internal-testing.

Conclusion

While in the domain of ML we have the dataset, the model and the tokenizer - each of which can be made tiny and enable super-speed development with low resource requirements, if you're coming from a different industry you can adapt the ideas discussed in this chapter to your particular domain's artifacts/payloads.

Backup of all scripts in this chapter

Should the original scripts this chapter is pointing to disappear or the HF hub is down while you're reading this, here is the local back up of all of them.

note-to-self: to make the latest backup of files linked to in this chapter run:

```
perl -lne 'while (/(https.*?.py)\)/g) { $x=$1; $x=~s/blob/raw/; print qq[wget $x] }'
make-tiny-models.md
```

Reducing the number of layers for large models

When debugging PyTorch workflows, as explained in using small payload you'd normally try to use tiny random models (as covered above). But since some problems only appear at scale it's very likely you'd have to use the full-sized model, which may take a very long time to load and run until it gets to the point of interest, where problems appear.

Given the nature of ML model architectures, they typically use a sequence of identical layers that repeat one after another. Therefore, if a model has, say, 48 layers, you can shrink it to just 2 layers, which will dramatically speed up both the loading and running the code. Of course, the qualitative outcome will be bad, but we aren't concerned with quality if the workload hangs or breaks.

Therefore in this section we will discuss how to reduce the model's number of hidden layers from many to just 1-2. If the layers aren't identical (e.g. some MoE models alternate between 2 different block configurations) then ensure you include at least one variation of each. For the purpose of the following demonstrations we will use this MoE model Qwen/Qwen3-30B-A3B-Instruct-2507. We have 48 hidden layers there as can be seen from its config file.

This model may have 2 alternating types of Transformer blocks, so we need to keep at least 2 layers. (`Qwen/Qwen3-Next-80B-A3B-Instruct` uses a full attention block only once every 4 layers so there you'd need at least 4 layers.)

The config entry that we want to change is `num_hidden_layers`. Some architectures pair it with a companion entry that has to be trimmed in lockstep - a decoder-side layer count, or a `layer_types` list - see Making a tiny model.

Let's first run a quick test to demonstrate that even just the model loading time can be much faster, before seeing the huge speedup in the compute time:

```
git clone https://huggingface.co/Qwen/Qwen3-30B-A3B-Instruct-2507
time python -c 'import sys; from transformers import AutoModelForCausalLM; \
AutoModelForCausalLM.from_pretrained(sys.argv[1])' ./Qwen3-30B-A3B-Instruct-2507
perl -pi -e 's|"num_hidden_layers": 48|"num_hidden_layers": 2|' Qwen3-30B-A3B-Instruct-2507/
config.json
time python -c 'import sys; from transformers import AutoModelForCausalLM; \
AutoModelForCausalLM.from_pretrained(sys.argv[1])' ./Qwen3-30B-A3B-Instruct-2507
```

so here we clone the model locally and then measured how long it took to load the base model:

```
real    5m59.857s
user    128m28.088s
sys     16m33.861s
```

then we reduced the number of layers from 48 to 2 and repeated the model loading. This time we get:

```
real    0m20.398s
user    2m9.101s
sys     2m29.587s
```

Looking at the `real` entry (wallclock time) we have 6 minutes loading for the full model vs 20 seconds for the shrunk

2-layer model - that's 18x times faster and ~5.5 minutes of waiting time saved! `real` is what matters for this kind of benchmark, since it's the waiting you actually do; the `user` and `sys` figures dwarf it because they count CPU time summed over every core that was busy, not elapsed time - time later in this chapter explains all three.

There are 3 ways to accomplish that.

In this discussion we presume you're using HF Transformers-based models, but the same methodology could be translated to other modeling frameworks.

1. local clone with config edits

After finding the desired model on https://huggingface.co/, clone its git repo to the local disk, modify the `num_hidden_layers` entry in `config.json`, and then load the model from the local clone (same as we have just shown when measuring model loading time).

```
git clone https://huggingface.co/Qwen/Qwen3-30B-A3B-Instruct-2507
perl -pi -e 's|"num_hidden_layers": 48|"num_hidden_layers": 2|' Qwen3-30B-A3B-Instruct-2507/
config.json
python -c 'import sys; from transformers import AutoModelForCausalLM; \
AutoModelForCausalLM.from_pretrained(sys.argv[1])' ./Qwen3-30B-A3B-Instruct-2507
```

Please make sure that you load the locally cloned version, that is:

```
- ... from_pretrained("Qwen/Qwen3-30B-A3B-Instruct-2507")
+ ... from_pretrained("./Qwen3-30B-A3B-Instruct-2507")
```

This approach is useful since you don't need to change the user-end code.

2. editing the config object on the fly

The other even simpler approach is to hack the config object on the fly. This requires no local cloning and is probably the easiest solution, though it requires modifying the end user code:

```
python -c 'import sys; from transformers import AutoModelForCausalLM, AutoConfig; \
c=AutoConfig.from_pretrained(sys.argv[1]); c.num_hidden_layers=2; \
m=AutoModelForCausalLM.from_pretrained(sys.argv[1], config=c)' Qwen/Qwen3-30B-A3B-Instruct-2507
```

And since you will end up with an incomplete model which will generate random outputs anyway, you can also save the overhead of loading the original model weights and just create the model on the fly like so:

```
python -c 'import sys; from transformers import AutoModelForCausalLM, AutoConfig; \
c=AutoConfig.from_pretrained(sys.argv[1]); c.num_hidden_layers=2; \
m=AutoModelForCausalLM.from_config(c)' Qwen/Qwen3-30B-A3B-Instruct-2507
```

3. hacking the architecture modeling code

This approach is most useful if you need to deal with multiple models of the same architecture and you don't want to modify the end user code.

First we clone HF Transformers and install its editable version:

```
git clone https://github.com/huggingface/transformers/tree/main/src/transformers
cd transformers
pip install -e .[dev]
```

Now we can tweak the code under `src/transformers` and it will be immediately visible to the Python environment that is being used.

Continuing the example of working with `Qwen/Qwen3-30B-A3B-Instruct-2507` model, we find the place where its architecture modeling code lives in the HF Transformers code base. For example, we can look at the `architectures` field in the model's config, which gives us `Qwen3MoeForCausalLM`. We now find the Python module where it lives in the HF Transformers code base:

```
$ grep -Ir "class Qwen3MoeForCausalLM" src/transformers/models
src/transformers/models/qwen3_moe/modeling_qwen3_moe.py:class
Qwen3MoeForCausalLM(Qwen3MoePreTrainedModel, GenerationMixin):
src/transformers/models/qwen3_moe/modular_qwen3_moe.py:class
Qwen3MoeForCausalLM(MixtralForCausalLM):
```

So we know it's in `src/transformers/models/qwen3_moe/modeling_qwen3_moe.py` (we don't care for `modular_qwen3_moe.py` in this situation, since it's `modeling_qwen3_moe.py` that gets loaded).

Now we open `src/transformers/models/qwen3_moe/modeling_qwen3_moe.py` in the editor and search for `num_hidden_layers` usages to find where the layers are initialized, which in this case is here:

```
class Qwen3MoeModel(Qwen3MoePreTrainedModel):
    def __init__(self, config: Qwen3MoeConfig):
        super().__init__(config)
        [...]
        self.layers = nn.ModuleList(
            [Qwen3MoeDecoderLayer(config, layer_idx) for layer_idx in
range(config.num_hidden_layers)]
        )
```

So now we just hack the value of `num_hidden_layers` and we are done:

```
class Qwen3MoeModel(Qwen3MoePreTrainedModel):
    def __init__(self, config: Qwen3MoeConfig):
        super().__init__(config)
        [...]
        config.num_hidden_layers = 2
        self.layers = nn.ModuleList(
            [Qwen3MoeDecoderLayer(config, layer_idx) for layer_idx in
range(config.num_hidden_layers)]
        )
```

and now as long as this version of HF Transformers is devel-installed (`pip install -e .`) into the running Python environment `Qwen3MoeForCausalLM`-type of models will only use the first 2 layers as if it were the full model.

If you need to load the full model, but only run a few layers, then you can hack the loop over the layers in the model's `forward`. If the original code in `Qwen3MoeModel.forward` was:

```
for decoder_layer in self.layers[: self.config.num_hidden_layers]):
    hidden_states = decoder_layer(...)
```

you can change to:

```
KEEP_N_LAYERS = 2
for idx, decoder_layer in enumerate(self.layers[: self.config.num_hidden_layers]):
    # note: shortcut for much faster completion
    if idx+1 > KEEP_N_LAYERS: continue
    hidden_states = decoder_layer(...)
```

Additional Notes

When you load a pre-trained model while shortening its layers stack, you're going to see a flurry of warnings telling you that some weights have been ignored.

Also I'm reminding that you will end up with a model which will allow you to perform functional checks and tune ups - memory usage and performance, etc. It will produce garbage and if you measure the loss it'll be very high (though it shouldn't be `NaN`).

You can now measure performance with say 2 and 4 layers and tell how much overhead each layer takes from the difference and extrapolate this to what the full model will need.

Memory usage-wise, unless there is a memory leak, after the first layer finished running, subsequent layers shouldn't consume any additional CPU or GPU memory (other than peak memory) if activation checkpointing is not used and `torch.cuda` memory cache isn't flushed. If activation checkpointing is enabled, then expect each layer to consume the same additional amount of memory as the previous one (of the size of the checkpointed tensor).

Other shrink-the-stack use cases

You can apply a similar hack to other components that also have stacks of identical code-wise blocks. For example, you could reduce the number of attention heads and then the attention mechanism will run much faster (but of course producing garbage, which is fine most of the time when we focus on functional debugging) or skipping most attention blocks completely.

When I was debugging 15M sequence length training using ALST - I would only run self-attention in the last layer and then skip it in the previous layers - this reduced my testing time from hours to minutes, since very long sequence length using full self-attention has an O(2) quadratic nature with regards to sequence length it attends to.

Let's say we run only attention in the last layer:

In attention `__init__` we set a few flags, let's use `Qwen3MoeAttention`:

```
    def __init__(self, config: Qwen3MoeConfig, layer_idx: int):
        super().__init__()
        self.skip_all_but_last_attention_debug_mode = True
```

```
        self.rotating_layer_counter = 0
```

and then in `Qwen3MoeAttention.forward`, we replace:

```
attn_output, attn_weights = attention_interface((self, query_states, ...)
```

(note the `...` - most args were trimmed for this exemplification), with:

```
import einops
if not self.skip_all_but_last_attention_debug_mode:
    attn_output, attn_weights = attention_interface(self, query_states, ...)
else:
    self.rotating_layer_counter = (self.rotating_layer_counter + 1) % self.num_hidden_layers
    # we detect the last layer by module counting since we know how many layers there are
    if self.rotating_layer_counter % self.num_hidden_layers == 0:
        attn_output, attn_weights = attention_interface(self, query_states, ...)
    else:
        # this feeds bogus data of the right shape connected to a graph - good enough for debug
        attn_output = einops.rearrange(query_states, "bs hc sl ... -> bs sl hcl ...")
        attn_weights = None
```

and, of course, install `pip install einops` for the above code to work.

Memory usage

GPU memory is probably the most invaluable resource, often more important than the compute and we always need more of it. So it's good to know how not to waste it.

The one error we all want to avoid is Out of Memory (aka CUDA OOM):

```
CUDA out of memory. Tried to allocate 15.41 GiB. GPU 3 has a total capacity of 79.10 GiB of which
15.40 GiB is free
```

Debugging CUDA OOM in `forward`

Let's first demo a simple CUDA OOM situation. Let's create a simple model and do a single `forward`/`backward` call on it:

```
import torch
class MyModel(torch.nn.Module):
    def forward(self, x):
        y = x.div(2)
        z = y.add(10)
        l = z.mul(2.0)
        return l.sum()

x = torch.rand(int(15e9), device="cuda", requires_grad=True) # 15B elements
```

```
net = MyModel()

loss = net(x) # implicit forward
loss.backward()
```

On NVIDIA B200 w/ 180GiB of HBM memory the above script will OOM:

```
$ python mem-oom.py
Traceback (most recent call last):
  File "/base/mem-oom.py", line 12, in <module>
    loss = net(x)
           ^^^^^^
  File "/home/stas/miniconda3/envs/dev/lib/python3.12/site-packages/torch/nn/modules/module.py",
line 1775, in _wrapped_call_impl
    return self._call_impl(*args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/home/stas/miniconda3/envs/dev/lib/python3.12/site-packages/torch/nn/modules/module.py",
line 1786, in _call_impl
    return forward_call(*args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/base/mem-oom.py", line 6, in forward
    l = z.mul(2.0)
        ^^^^^^^^^^
torch.OutOfMemoryError: CUDA out of memory. Tried to allocate 55.88 GiB. GPU 0 has a total
capacity of 178.35 GiB of which 10.00 GiB is free. Including non-PyTorch memory, this process has
168.34 GiB memory in use. Of the allocated memory 167.64 GiB is allocated by PyTorch, and 4.62
MiB is reserved by PyTorch but unallocated. ...
```

The traceback tells us where exactly the problem is:

```
  File "/base/mem-oom.py", line 6, in forward
    l = z.mul(2.0)
```

Now we can easily act on it, e.g., reducing sequence length, batch size, choosing a smaller model or possibly improving the code to be more memory efficient.

footnote: if your GPU is of a different size you will need to tweak the `x` to be larger or smaller.

Debugging CUDA OOM in `backward`

Now what happens if the CUDA OOM happens in `backward`?

Unless one writes their own `torch.autograd` class, PyTorch will autogenerate the `backward` function for each corresponding `forward` it runs. And since it is autogenerated, whenever there is an issue in `backward` it becomes very difficult to find what `forward` call (and the exact line number) is responsible for it and thus understand the cause of the problem.

It's quite tricky to write a simple script to trigger a CUDA OOM in an auto-generated `backward`, because when `backward` starts, PyTorch immediately drops the temporary tensors from `forward`, freeing memory for `backward` memory allocations to almost always be successful.

Therefore, let's modify the script from the previous section to intentionally create a temporary memory leak, by returning the intermediate tensors from `forward`, so that their reference counter doesn't go to 0 and they don't get reclaimed when `backward` starts, thus not freeing the CUDA memory they hold.

And let's shrink `x` to 10B elements so that `forward` doesn't OOM, but consumes most of the memory leading to a CUDA OOM in `backward`.

```
import torch
class MyModel(torch.nn.Module):
    def forward(self, x):
        y = x.div(2)
        z = y.add(10)
        l = z.mul(2.0)
        return l.sum(), y, z, l

x = torch.rand(int(10e9), device="cuda", requires_grad=True) # 10B elements
net = MyModel().cuda()

loss, y, z, l = net(x)
loss.backward()
```

Let's run it:

```
$ python mem-oom.py
Traceback (most recent call last):
  File "/base/mem-oom.py", line 13, in <module>
    loss.backward()
  File "/home/stas/miniconda3/envs/dev/lib/python3.12/site-packages/torch/_tensor.py", line 625,
in backward
    torch.autograd.backward(
  File "/home/stas/miniconda3/envs/dev/lib/python3.12/site-packages/torch/autograd/__init__.py",
line 354, in backward
    _engine_run_backward(
  File "/home/stas/miniconda3/envs/dev/lib/python3.12/site-packages/torch/autograd/graph.py",
line 841, in _engine_run_backward
    return Variable._execution_engine.run_backward(  # Calls into the C++ engine to run the
backward pass

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
torch.OutOfMemoryError: CUDA out of memory. Tried to allocate 37.25 GiB. GPU 0 has a total
capacity of 178.35 GiB of which 28.63 GiB is free. Including non-PyTorch memory, this process has
149.71 GiB memory in use. Of the allocated memory 149.01 GiB is allocated by PyTorch, and 6.11
MiB is reserved by PyTorch but unallocated. ...
```

As you can see this traceback tells us absolutely nothing about where the problem is and thus in a real world, where there could be dozens of classes and layers in the model, it's quite impossible to identify the corresponding `forward` call other than perhaps via some smart bisection of the `forward` calls (short-circuiting a few at a time), which would be quite difficult and time consuming and error-prone.

Luckily, PyTorch has a special autograd mechanism that is used to detect anomalies. When enabled it'll augment the `backward` traceback with the `forward` traceback corresponding to the `backward` code.

Let's move the implicit `forward` and explicit `backward` calls into the `torch.autograd.detect_anomaly()` context manager:

```
with torch.autograd.detect_anomaly():
    loss, y, z, l = net(x)
    loss.backward()
```

with the complete code now becoming:

```
import torch
class MyModel(torch.nn.Module):
    def forward(self, x):
        y = x.div(2)
        z = y.add(10)
        l = z.mul(2.0)
        return l.sum(), y, z, l

x = torch.rand(int(10e9), device="cuda", requires_grad=True) # 10B elements
net = MyModel().cuda()

with torch.autograd.detect_anomaly():
    loss, y, z, l = net(x)
    loss.backward()
```

footnote: it's crucial that both the `forward` and the `backward` calls are inside this context manager.

Let's run this updated script:

```
$ python mem-oom.py
/base/mem-oom.py:12: UserWarning: Anomaly Detection has been enabled. This mode will increase the
runtime and should only be enabled for debugging.
  with torch.autograd.detect_anomaly():
/home/stas/miniconda3/envs/dev/lib/python3.12/site-packages/torch/autograd/graph.py:841:
UserWarning: Error detected in MulBackward0. Traceback of forward call that caused the error:
  File "/base/mem-oom.py", line 13, in <module>
    loss, y, z, l = net(x)
  File "/home/stas/miniconda3/envs/dev/lib/python3.12/site-packages/torch/nn/modules/module.py",
line 1775, in _wrapped_call_impl
    return self._call_impl(*args, **kwargs)
  File "/home/stas/miniconda3/envs/dev/lib/python3.12/site-packages/torch/nn/modules/module.py",
line 1786, in _call_impl
    return forward_call(*args, **kwargs)
  File "/base/mem-oom.py", line 6, in forward
    l = z.mul(2.0)
 (Triggered internally at /pytorch/torch/csrc/autograd/python_anomaly_mode.cpp:122.)
  return Variable._execution_engine.run_backward(  # Calls into the C++ engine to run the
```

```
backward pass
Traceback (most recent call last):
  File "/base/mem-oom.py", line 14, in <module>
    loss.backward()
  File "/home/stas/miniconda3/envs/dev/lib/python3.12/site-packages/torch/_tensor.py", line 625,
in backward
    torch.autograd.backward(
  File "/home/stas/miniconda3/envs/dev/lib/python3.12/site-packages/torch/autograd/__init__.py",
line 354, in backward
    _engine_run_backward(
  File "/home/stas/miniconda3/envs/dev/lib/python3.12/site-packages/torch/autograd/graph.py",
line 841, in _engine_run_backward
    return Variable._execution_engine.run_backward(  # Calls into the C++ engine to run the
backward pass

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
torch.OutOfMemoryError: CUDA out of memory. Tried to allocate 37.25 GiB. [...]
```

Now in addition to the original traceback, we also got the corresponding `forward` traceback:

```
Traceback of forward call that caused the error:
  File "/base/mem-oom.py", line 13, in <module>
    loss, y, z, l = net(x)
  File "/home/stas/miniconda3/envs/dev/lib/python3.12/site-packages/torch/nn/modules/module.py",
line 1775, in _wrapped_call_impl
    return self._call_impl(*args, **kwargs)
  File "/home/stas/miniconda3/envs/dev/lib/python3.12/site-packages/torch/nn/modules/module.py",
line 1786, in _call_impl
    return forward_call(*args, **kwargs)
  File "/base/mem-oom.py", line 6, in forward
    l = z.mul(2.0)
 (Triggered internally at /pytorch/torch/csrc/autograd/python_anomaly_mode.cpp:122.)
```

and in particular we narrow it down to this section:

```
  File "/base/mem-oom.py", line 6, in forward
    l = z.mul(2.0)
```

So now we know exactly where in the `forward` code there is an issue that gets triggered in the `backward` and now we can start working on fixing it.

Besides using a context manager, you can also activate it globally somewhere before the training loop using `torch.autograd.set_detect_anomaly(True)`.

Additionally you can disable the `NaN` checker to make things a bit faster with: `torch.autograd.set_detect_anomaly(True, check_nan=False)`.

Important: make sure to disable the anomaly detection mode before you put your work to real use because it will slow things down.

Overcoming CUDA OOM due to memory fragmentation

When developing Arctic Long Sequence Training a lot of the work involved hard (`contiguous`) reshaping of tensors for Sequence Parallelism, which lead to a very poor HBM utilization, because the reshapes lead to memory fragmentation.

The solution was to use:

```
export PYTORCH_ALLOC_CONF=expandable_segments:True
```

which made a dramatic positive impact to allowing for a much longer sequence length to be used w/o incurring a CUDA OOM event.

In the particular case of this project I haven't observed any noticeable performance degradation, but if you use it do benchmark the performance w/ and w/o it to ensure it doesn't impact your workload's performance for the worse.

footnote: the original env var name was `PYTORCH_CUDA_ALLOC_CONF`, but it got renamed in recent PyTorch versions.

Discovering how many GBs is allocatable before OOM for CPU and GPU

Here are 2 simple one-liners that can tell you how much memory you can allocate on cpu and gpu. We will be using an H200 node with ~2TiB of CPU RAM and GPUs of 144GiB of HBM memory.

On CPU:

```
$ python -c 'import torch; [(torch.ones((1024*2**18)), print(c)) for c in range(2000)]'
0
1
2
[...]
1996
1997
Killed
```

This one liner tried to allocate 2TiB of memory, 1GiB at a time, reporting each successful incremental allocation. We can see the program incurred cpu-oom after successfully allocating around 1997GiB.

If your admin set `cgroups` to cpu oom individual programs when a collective amount of cpu memory used by a given user reaches a specific size, this is how you can discover what that value is. For example, on a shared 8-GPU node with 2TiB of CPU RAM, and you asked for just 1x GPU - you will likely get 1/8th of node's total resources - thus any of your processes may get killed via cpu-oom at 250GiB (`2000/8`), even though `top` shows you all of 2TiB available.

Now let's do the same test on GPU (after moving tensors to `cuda` device):

```
$ python -c 'import torch; [(torch.ones((1024*2**18)).cuda().contiguous(), print(c)) for c in
range(200)]'
0
1
2
3
[...]
137
```

```
138
Traceback (most recent call last):
  File "<string>", line 1, in <module>
torch.OutOfMemoryError: CUDA out of memory. Tried to allocate 1024.00 MiB.
GPU 0 has a total capacity of 139.80 GiB of which 289.25 MiB is free. [...]]
```

Indeed, H200 is about 140GiB, so this checks out. You might wonder what's the point of this. This discovery code is useful when you have to deal with multiple systems accessing the same GPU - e.g. VLLM and VeRL sharing the same GPUs and PyTorch memory usage counters are completely wrong, or if you want to emulate a smaller GPU memory while using a larger GPU (use case: you have to make your hparams fit into H200 but you lost access to it and you have a larger B300 GPU - just use up that many GiB and now your B300 is like H200 memory-size-wise)

Both of these one-liners will be useful in the investigation process of following section.

Overcoming the coherent memory uncertain behavior

The introduction of coherent/unified memory in recent NVIDIA products like DGX Station, Spark, GH200, GB300 makes it for a very confusing accounting, hard to understand memory usage patterns and unstable behavior.

For example, with DGX Spark, you can't even get a memory reading from the GPU side - `nvidia-smi` will not report memory use and will say `Not Supported` and `pynvml.nvmlDeviceGetMemoryInfo()` will assert. Here you never know if a CPU RAM using program is going to steal memory from a GPU program and OOM it (or vice versa). The same GPU workload may work at one time and OOM at another depending on what's running on the CPU side.

At least in my latest experiments with DGX Station here is what appears to be true. With the default settings Linux may borrow from the GPU HBM memory if it runs out of its LPDDR5 memory, but not the other way around - since PyTorch doesn't implement borrowing from CPU RAM as of 2026-08.

We will use the 2 one-liners from the previous section to see what's possible here. We will be using a DGX Station with 277GiB of GPU HBM memory and 496GiB of LPDDR5 CPU RAM - the total for the coherent memory reported by `top` is about 770GiB.

On CPU:

```
python -c 'import torch; [(torch.ones((1024*2**18)), print(c)) for c in range(900)]'
```

This program will try to allocate 900GiB of memory, one GiB at a time, reporting each successful incremental allocation. So on DGX Station when GPU is not used we get:

```
$ python -c 'import torch; [(torch.ones((1024*2**18)), print(c)) for c in range(900)]'
0
1
2
3
...
748
749
750
Killed
```

The program CPU OOMed at ~750GiB, since some 20GiB were used by system programs. So now we know if nothing else

runs we can use all of the coherent memory - once the CPU RAM is fully consumed it'll start using GPU's HBM memory. (However note that `nvidia-smi` / NVML will not report this usage!)

Now let's check how much can we allocate on GPU:

```
$ python -c 'import torch; [(torch.ones((1024*2**18)).cuda().contiguous(), print(c)) for c in
range(500)]'
0
1
2
3
[...]
272
273
274
Traceback (most recent call last):
  File "<string>", line 1, in <module>
torch.OutOfMemoryError: CUDA out of memory. Tried to allocate 1024.00 MiB.
GPU 0 has a total capacity of 276.50 GiB of which 797.44 MiB is free. [...]]
```

OK, so we can see PyTorch (at least of this writing) won't reach into CPU RAM and has a hard stop at the size of HBM.

case study: As I was trying to figure out the maximum sequence length I could do post-training with on DGX Station with Qwen3-32B model, I kept getting alternating CPU-OOM and GPU-OOM events. The reason was that I was offloading optimizer states to CPU RAM since there wasn't enough HBM memory to hold the weights and optimizer states on a single GB300 GPU, but since there isn't that much CPU RAM either, Linux was reaching into the GPU "cookie jar" and stealing GPU memory. It was a very unsettling experience since I had to run dozens of experiments, most of them leading to one or the other OOM, why? because depending on what CPU-side processes were running (think various Linux daemons and user programs) there would be a different amount of CPU RAM available at different times and the behavior would become unpredictable.

The solution: to prevent CPU from stealing GPU memory switch to the CDMM mode (from the default NUMA mode), as explained in a paper linked from this post by running:

```
$ echo options nvidia NVreg_CoherentGPUMemoryMode=driver | sudo tee /etc/modprobe.d/
nvidia-openrm.conf
```

and rebooting. When the system is back test it's set correctly:

```
$ grep Coherent /proc/driver/nvidia/params
CoherentGPUMemoryMode: "driver"
```

If it's not "driver" then it didn't work.

Voila, now you're back into the reliable and predictable "CPU memory is CPU memory, GPU memory is GPU memory" world. No weird surprises and various memory counters reflect reality.

If you rerun the first one-liner after switching to the CDMM mode, you will now see:

```
$ python -c 'import torch; [(torch.ones((1024*2**18)), print(c)) for c in range(900)]'
0
1
...
475
Killed
```

Since there is only 496GiB of LPDDR5 and some 20GiB are used by system processes we can now clearly see that CPU wasn't allowed to reach into GPU memory (I wasn't using GPU while running this test - its memory was fully available.) If you remember, originally the one-liner reported the process being CPU-OOMed at ~750GiB.

In the case of DGX Spark where it's only 120GiB of LPDDR5 memory, shared between CPU and GPU and 0 HBM memory, ideally the developer should be able to assign how much of the memory should go to GPU and to CPU - e.g. 80 and 40 correspondingly - that way the developer can reliably plan their workload and avoid programs crashing. I communicated this need to the NVIDIA team, let's see if they will come back with a solution to us.

PyTorch memory profiler

PyTorch memory profiler is quite easy to use. It requires 2 stages.

Stage 1. Instrument and run the code under `torch.cuda.memory` profiler

```
import torch
torch.cuda.memory._record_memory_history(max_entries=int(1e9))
# your tensor creation code goes here, e.g.:
t = torch.zeros(100,100, device="cuda")
torch.cuda.memory._dump_snapshot("/tmp/mem.pickle")
```

Here we just allocate a small tensor of zeros.

If you're on a multi-gpu setup you'd want to write a profile dump per rank to an individual file:

```
rank = torch.distributed.get_rank() # assuming torch.dist has already been initialized
torch.cuda.memory._dump_snapshot(f"/tmp/mem-{rank}.pickle")
```

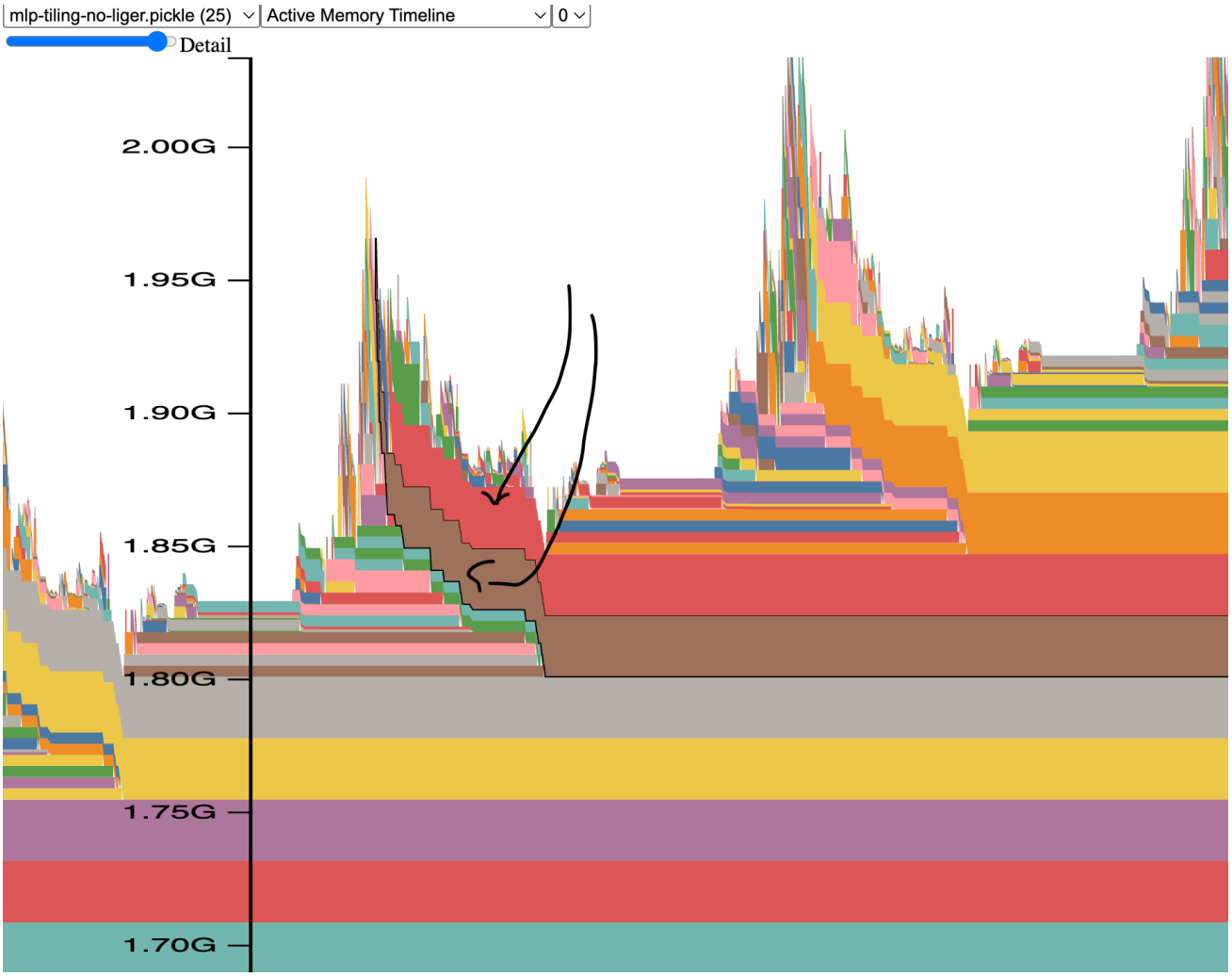
or just save one rank instead if everything is symmetrical. The first time I missed this nuance and I was getting weird results, since I was hitting a race condition of different ranks writing to the same file, which looked non-corrupt when rendered but the outcome was a big mess.

For a largish code you would want to record as many memory allocation/free events as possible so I normally use a pretty large value like `max_entries=int(1e9)` for the `_record_memory_history` call.

Stage 2. Render the saved profile information into a visual representation

At this stage you'd typically go to https://docs.pytorch.org/memory_viz and drop the memory profile pickle file you generated in Stage 1 into the browser at that URL. The most useful feature there is the 'Active memory profile` drop-down menu.

To get a feeling for what it looks like, here is an example of a memory profile rendering for a memory leak I discovered while I was working on a tricky implementation of a `TiledMLP torch.autograd.Function`.



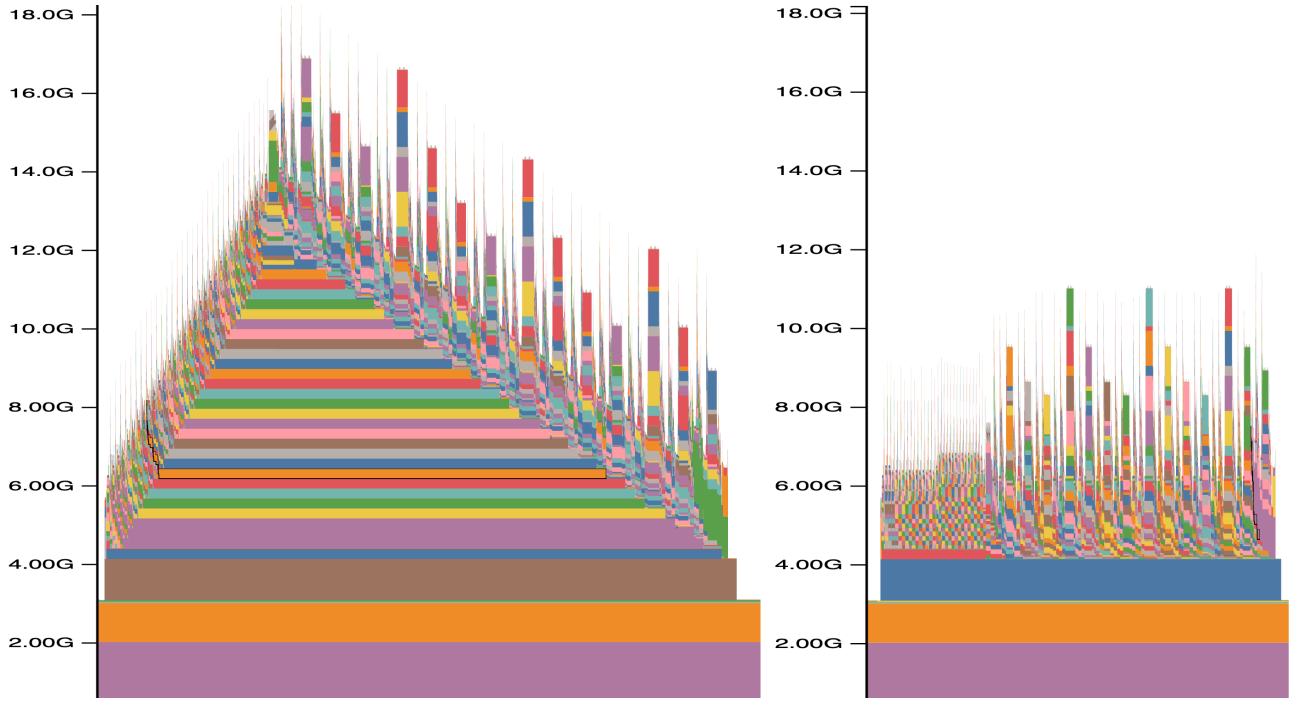
You can see those brown- and red-colored continuous horizontal bars (I pointed to those with black arrows). On the very left edge of those bars are the moments that created 2 large tensors during a single layer's `forward`, but you can see those 2 unlike other colored bars continue all the way into the right edge. The exact same story happen in the next spike, which is just the subsequent layer's memory allocations when it runs its `forward` - and you can see the yellow and orange bars that demonstrate the same leak, because it doesn't get cleared. So each layer's `forward` here leaks a few MBs of memory, which quickly adds up. A very small model has been used here, so that the absolute leak size was small, but once switched to a real model those MBs become GBs and we quickly run out of memory.

You can click on all those bars and the profiler will show you the traceback to the code that created the corresponding memory allocation. Since under the hood, PyTorch runs C++ CUDA code, unless you understand what happens there, it won't help you to understand the location of the leak in the code. But if you trace back up the trace into the Python land, you will actually see references to functions that you'd be familiar with. For example, calls like `torch.zeros()`.

As long as you're in the `forward` function it's relatively easy to find where in the code leaks comes from. But if it's a `backward` it becomes much more complicated unless you're debugging a custom autograd function. Still it should give you enough information to be able to ask for help if you can't figure it out yourself. The best recourse in that situation is to try to reduce the code to the minimal size reproducible Python script that others can reproduce the problem with and then ask at some place where PyTorch developers hang out - for example I find `#questions` at the PyTorch Slack workspace to be an invaluable resource. If you don't have access to that Slack workspace, fear not, https://discuss.pytorch.org/ should work just as well. Even better, using the latter will help others to find answers to the same question down the road.

Besides profiling memory leaks, this functionality is also useful for showing how different implementations of the same algorithm use a different amount of gpu memory. For example, the following visualization I prepared for the Arctic Long

Sequence Training paper:



This visualization depicts a PyTorch memory profile over a single `forward-backward` iteration. Left: normal setup. Right: with activation checkpoint offloading to CPU enabled.

The left side visualization is very telling to how gpu memory is used in the `forward` and `backward` calls, you can see how the left side, which is about 1/3 of the plot, is the layer-by-layer `forward` calls, and the right side, which is the remaining 2/3 of the plot, is layer-by-layer `backward` calls. You can also see that `backward` takes 2x longer than `forward` because it has do to compute gradients wrt weights and inputs. You can also see that typically `forward` allocates a lot of memory, which `backward` then gradually releases, while also doing small allocations of its own.

The right side visualization shows how very different memory usage pattern is, if we don't store any intermediary tensors on the gpu and offload them to cpu memory. You can see the same `forward` and `backward` calls but now the memory plot is flat, so you can add many more layers and it'll still use the same amount of gpu memory, whereas the image on left shows that if there are too many layers one will run into OOM, as the hill will continue to climb. If you're curious, the big spikes during `backward` in both images are gradient reductions across gpus.

This shows that even if you don't suspect a memory leak in your code it might still be a good idea to run it through memory profiler and you might get ideas to how to reduce memory usage or at the very least you will have a better feel for what your code is doing with the gpu memory.

Additional important notes:

- Try to limit the profiler dump to just a few iterations, otherwise when you try to render the results in the browser it's likely to crash. You always want at least 2 iterations since the first one is always an outlier. I usually do 3 iterations.

Additional resources:

- Understanding GPU Memory 1: Visualizing All Allocations over Time
- HF folks made an improved rendering version.

Strategic memory allocation tracing

While external memory profilers can be very useful, often having control over when you take a sample of GPU and CPU memory usage is needed. `see-mem-usage` debug util has been developed by the DeepSpeed project and I made some small

tweaks to it:

see-mem-usage.py

You want to make sure `pip install nvidia-ml-py` is run once, so that the report includes not only the CUDA memory report but the total gpu memory usage, since CUDA memory allocator is not always used. e.g., NCCL memory allocations aren't visible by CUDA and thus aren't accounted for, but can consume GBs of gpu memory. The total memory usage identical to what `nvidia-smi` reports is the `NV` column in the report.

A critical nuance when tracing GPU memory usage is that if you released a python variable containing a tensor it doesn't necessarily mean the tensor gets immediately freed. Python's garbage collection is run on a schedule and thus it's critical to run `gc.collect()` after releasing critical large environment variables (while debugging!) and only then sampling memory usage, which is what this library does for you behind the scenes.

Needless to say you will not want to use this library in production, since the overhead of frequent `gc.collect` calls and nvml sampling adds a non-trivial runtime overhead. So remember to flip `force=True` to `force=False` and then you can leave the debug code in your production code if desired.

So let's run a little program that allocates a tensor, copies it to cpu, frees it on gpu and then frees the cpu copy.

```
    device = "cuda" if torch.cuda.is_available() else "cpu"
    see_memory_usage("before alloc", force=True)
    t1 = torch.zeros(100_000,10_000, device=device)
    t2 = torch.zeros(100_000,10_000, device=device)
    del t2
    see_memory_usage("after alloc", force=True)
    c1 = t1.cpu()
    see_memory_usage("after copy to cpu", force=True)
    del t1
    see_memory_usage("after freeing on gpu", force=True)
    del c1
    see_memory_usage("after freeing on cpu", force=True)
```

Let's look at the output. The above program is at the bottom of the `see-mem-usage.py` library)

```
$ python see-mem-usage.py
[0] mp: before alloc
[0] mp: MA 0.00 GiB | Max_MA 0.00 GiB | CA 0.00 GiB | Max_CA 0.00 GiB | NV 0.71 GiB | CPU mem:
proc 0.51 GiB / node 84.09 GiB (4.2%)
[0] mp: after alloc
[0] mp: MA 3.73 GiB | Max_MA 7.45 GiB | CA 7.45 GiB | Max_CA 7.45 GiB | NV 8.87 GiB | CPU mem:
proc 0.70 GiB / node 84.26 GiB (4.2%)
[0] mp: after copy to cpu
[0] mp: MA 3.73 GiB | Max_MA 3.73 GiB | CA 7.45 GiB | Max_CA 7.45 GiB | NV 8.87 GiB | CPU mem:
proc 4.42 GiB / node 88.02 GiB (4.4%)
[0] mp: after freeing on gpu
[0] mp: MA 0.00 GiB | Max_MA 3.73 GiB | CA 7.45 GiB | Max_CA 7.45 GiB | NV 8.87 GiB | CPU mem:
proc 4.42 GiB / node 88.01 GiB (4.4%)
[0] mp: after freeing on cpu
[0] mp: MA 0.00 GiB | Max_MA 0.00 GiB | CA 7.45 GiB | Max_CA 7.45 GiB | NV 8.87 GiB | CPU mem:
proc 0.70 GiB / node 85.41 GiB (4.3%)
```

Legend:

- `MA`: `torch.cuda.memory_allocated()` - how much memory has been allocated at this moment
- `Max_MA`: `torch.cuda.max_memory_allocated()` - what was the peak memory usage so far (we also reset this counter after each run, so it will show the peak memory since the last call to `see_mem_usage`)
- `CA` : `torch.cuda.memory_reserved()`
- `Max_CA`: `torch.cuda.max_memory_reserved()`
- `NV`: current total memory usage like `nvidia-smi` report, which is almost always more than what's reported by torch.cuda (the `MA` column)
- `CPU mem`: two point-in-time CPU RAM readings from `psutil` - `proc` is *this* process's resident memory (RSS, `psutil.Process().memory_info().rss`), which is what you usually want when debugging your own program, and `node` is host-wide RAM in use (`total - available`) with its `percent` of total. On a shared node the `node` figure includes everyone else's memory, so watch `proc` for your process and use `node` only to see how close the whole box is to full. Both are snapshots, not peaks - for peak CPU usage see Getting program's CPU peak memory usage

Now that we know what each column stands for let's analyze the output of the program.

```
[0] mp: before alloc
[0] mp: MA 0.00 GiB | Max_MA 0.00 GiB | CA 0.00 GiB | Max_CA 0.00 GiB | NV 0.71 GiB | CPU mem:
proc 0.51 GiB / node 84.09 GiB (4.2%)
```

If you look at the `NV` column you can see the gpu was already using 0.71GiB of memory, even though no tensor has been allocated yet. This is because CUDA loads compute kernels the first time you call `import torch` - note that `torch.cuda` is not reporting that! all its columns are zeros. On the CPU side `proc` is 0.51GiB, which is just the resident memory of the Python process after `import torch`.

Then we execute:

```
    t1 = torch.zeros(100_000,10_000, device=device)
    t2 = torch.zeros(100_000,10_000, device=device)
    del t2
```

and the corresponding log around it is:

```
[0] mp: before alloc
[0] mp: MA 0.00 GiB | Max_MA 0.00 GiB | CA 0.00 GiB | Max_CA 0.00 GiB | NV 0.71 GiB | CPU mem:
proc 0.51 GiB / node 84.09 GiB (4.2%)
[0] mp: after alloc
[0] mp: MA 3.73 GiB | Max_MA 7.45 GiB | CA 7.45 GiB | Max_CA 7.45 GiB | NV 8.87 GiB | CPU mem:
proc 0.70 GiB / node 84.26 GiB (4.2%)
[0] mp: after copy to cpu
```

So we can see that `MA` is half of `Max_MA` - because we had 2 tensors of the same size allocated and one of them freed. So the CUDA peak memory of 7.45GiB is 2x larger than the current CUDA memory usage. This is a very important moment. Often the software OOMs exactly during peak memory usage. For example, if some intermediary tensor isn't freed up fast enough it could cause OOM - and also see the earlier note about python garbage collection, there are rare situations where a well placed `gc.collect` call can save the day and prevent OOM.

The `CA` and `MaxCA` columns report cached memory, I often find those not very useful for memory debug purposes, I sometimes even add:

```
torch.cuda.empty_cache()
```

to prevent caching getting in the way of accounting, but this one is definitely going to slow things down. The snippet is in `see_memory_usage`, but commented out.

But caching will lead to `nvidia-smi` or the `NV` column in this report to reporting cached memory. In the `after alloc` row above you can see that while `torch.cuda` reports only 3.73GiB of actively allocated memory (`MA`), `NV` is 8.87GiB, because some of the memory got cached (`CA` 7.45GiB), but even that doesn't fully check out.

`8.87-7.45=1.42` GiB, whereas the previous `see_mem_usage` before tensor allocation reported `NV` 0.71GiB, in other words some other gpu memory allocations that `torch.cuda` hasn't accounted for have happened and we have no idea what they are! Watch that delta between what CUDA columns and the NV column, sometimes you might find many GiBs are unaccounted for.

What happened here is most likely PyTorch `torch.zeros` call loaded some additional CUDA kernels which took another half GB of GPU memory (again unaccounted for). `torch.distributed` with NCCL is another large source of "lost" GPU memory.

Next, we copy one tensor to cpu memory:

```
    c1 = t1.cpu()
```

which gives us:

```
[0] mp: after alloc
[0] mp: MA 3.73 GiB | Max_MA 7.45 GiB | CA 7.45 GiB | Max_CA 7.45 GiB | NV 8.87 GiB | CPU mem:
proc 0.70 GiB / node 84.26 GiB (4.2%)
[0] mp: after copy to cpu
[0] mp: MA 3.73 GiB | Max_MA 3.73 GiB | CA 7.45 GiB | Max_CA 7.45 GiB | NV 8.87 GiB | CPU mem:
proc 4.42 GiB / node 88.02 GiB (4.4%)
```

we see the `torch.cuda` and NV counters remain the same but the CPU `proc` reading jumped from 0.70GiB to 4.42GiB - a delta of ~3.73GiB, exactly the size of the tensor we just copied to CPU (`100_000*10_000*4 bytes = 3.73GiB` in fp32). This is the payoff of tracking process RSS rather than host-wide memory: the delta is your program's own allocation, clean of whatever else is running on the node. The `node` column moved too (84.26 -> 88.02GiB), but it's noisier - it drifts with every other process on the box.

One other thing to observe here is that `MA 3.73 GiB | Max_MA 3.73 GiB` - current and peak memory usage are the same, since there were no memory allocations or freeing on gpu at this step.

Next we delete the remaining tensor on CUDA (`t1`):

```
[0] mp: after copy to cpu
[0] mp: MA 3.73 GiB | Max_MA 3.73 GiB | CA 7.45 GiB | Max_CA 7.45 GiB | NV 8.87 GiB | CPU mem:
proc 4.42 GiB / node 88.02 GiB (4.4%)
[0] mp: after freeing on gpu
```

```
[0] mp: MA 0.00 GiB | Max_MA 3.73 GiB | CA 7.45 GiB | Max_CA 7.45 GiB | NV 8.87 GiB | CPU mem:
proc 4.42 GiB / node 88.01 GiB (4.4%)
```

and we see that `MA` has gone to 0, which is what we would expect, CUDA no longer has any active tensors. Note that the peak memory isn't zero, since there was exactly the size of that tensor allocation since the last time that counter was reset in `see_mem_usage` call. The CPU `proc` reading stays at 4.42GiB - we freed the GPU tensor but the CPU copy `c1` is still resident.

Finally we free the tensor on cpu:

```
[0] mp: after freeing on gpu
[0] mp: MA 0.00 GiB | Max_MA 3.73 GiB | CA 7.45 GiB | Max_CA 7.45 GiB | NV 8.87 GiB | CPU mem:
proc 4.42 GiB / node 88.01 GiB (4.4%)
[0] mp: after freeing on cpu
[0] mp: MA 0.00 GiB | Max_MA 0.00 GiB | CA 7.45 GiB | Max_CA 7.45 GiB | NV 8.87 GiB | CPU mem:
proc 0.70 GiB / node 85.41 GiB (4.3%)
```

and now the CPU `proc` reading drops from 4.42GiB back to 0.70GiB - releasing `c1` returned exactly the ~3.73GiB it had taken, landing back at the post-`import` baseline. The `node` column also came down, though not all the way to its starting value, since it tracks the whole host and other processes moved in the meantime.

The CUDA memory caches are still there as can be seen from `CA` and `Max_CA` columns, and `NV` reflects that plus some other non-CUDA allocation as discussed earlier.

If at the very end we add:

```
torch.cuda.empty_cache()
see_memory_usage("after empty cache", force=True)
```

we would see:

```
[0] mp: after empty cache
[0] mp: MA 0.00 GiB | Max_MA 0.00 GiB | CA 0.00 GiB | Max_CA 7.45 GiB | NV 1.42 GiB | CPU mem:
proc 0.70 GiB / node 85.41 GiB (4.3%)
```

Note how the `CA` columns is now 0, `Max_CA` column is still non-zero because it was still reporting peak, if we call `see_memory_usage` yet another time, it'd go to 0 as well.

But the interesting other number here is `NV 1.42 GiB` which tells us that there was 1.42GiB of memory allocated outside of the purview of `torch.cuda` - and it matches the `NV - CA` delta of `8.87-7.45=1.42`GiB we spotted back in the `after alloc` row, now that the cache is emptied and no longer hiding it. When I try to debug memory leaks that are inside PyTorch that when I enable `torch.cuda.empty_cache()` inside `see_memory_usage` because then it reports the delta for me and I don't need to do any math.

You can't imagine how often I use this debug utility in my day-to-day work. Every so often I sprinkle these calls around the strategic places I suspect and start mapping out block by block and then narrowing down to the suspect areas. Foe example, one useful use case is to run this report before `forward`, `backward` and `step` and observe if each training iteration leaks a bit of memory and where:

```
    see_memory_usage("before fwd", force=True)
    output = model(**inputs)
    see_memory_usage("before bwd", force=True)
    output.loss.backward()
    see_memory_usage("before step", force=True)
    optimizer.step()
    see_memory_usage("after step", force=True)
```

For example, here is how I found a memory leak in `all_gather_object`, which you can see from this Issue. And there were several other similar leaks in PyTorch I discovered using this tool - all have been fixed since then. But more often, of course, the memory leaks are in my code ;)

CPU Memory

Debugging CPU memory OOM

This one is often very tricky to debug especially when a compute node is shared with others and each user gets to enjoy only a slice of the available CPU memory.

Once Resident cpu memory (RSS in `top`) hits the preset limit the program will get killed. There is no nice OOM message like we get with CUDA running out of memory, but you just get a single message:

```
Killed
```

which is very difficult to notice. This is typically performed by an `oom-kill` via cgroups. The `SIGKILL` is not trappable, so the program itself can't report anything - but the kernel does: see dmesg in the companion book for the entry naming the process it killed, how much memory it was holding and which limit it hit, plus the cgroup counters that record the kill without needing `sudo`.

note: Moreover in some situations, as in recent kubernetes implementations, the user gets kicked out from the job allocation, which makes it even more difficult to debug. Kubernetes Silent Pod Killer. This k8s "feature" makes no sense to me.

In the world of ML, you're likely to encounter this issue if you're doing massive parallel data preprocessing or you do GPU memory offloading to CPU memory. But more often when you build python wheels for massive packages like Flash Attention 2 w/o defining `MAX_JOBS` to be something quite small.

Getting program's CPU peak memory usage

One way was discussed in Strategic memory allocation tracing where you inject `see_memory_usage` during the program execution, but that's invasive and is not always easily doable, especially what if it's not a Python program that is causing the problem. Besides, its CPU `proc/node` columns are point-in-time snapshots, not peaks - only the GPU `Max_*` columns track peak usage (since the last reset), so unless a call happens to land exactly on the high-water mark it won't tell you the actual CPU peak.

So let's look at tools that report CPU peak memory usage, w/o needing to use full blown memory profiler.

/usr/bin/time

So the first program we look at is `/usr/bin/time`. Do not confuse it with the Bash's built-in `time`, which only reports runtime stats. Other shells beside Bash may have the built-in version as well.

Let's run an example:

```
$ /usr/bin/time -v python -c "import torch"
        Command being timed: "python -c import torch"
        User time (seconds): 8.12
        System time (seconds): 0.24
        Percent of CPU this job got: 629%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:01.33
        Average shared text size (kbytes): 0
        Average unshared data size (kbytes): 0
        Average stack size (kbytes): 0
        Average total size (kbytes): 0
        Maximum resident set size (kbytes): 640688
        Average resident set size (kbytes): 0
        Major (requiring I/O) page faults: 3
        Minor (reclaiming a frame) page faults: 75005
        Voluntary context switches: 489
        Involuntary context switches: 19
        Swaps: 0
        File system inputs: 0
        File system outputs: 8
        Socket messages sent: 0
        Socket messages received: 0
        Signals delivered: 0
        Page size (bytes): 4096
        Exit status: 0
```

While you can see that it does provide the same measurements as Bash's `time`:

```
        User time (seconds): 8.12
        System time (seconds): 0.24
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:01.33
```

What we want this time is this line:

```
        Maximum resident set size (kbytes): 640688
```

This gives us the peak memory used by the program, which is the highest amount of CPU memory the program used at any given point of its run. So if you measured your program needing let's say 200GiB of CPU RAM and then you try to run it elsewhere where you only have 132GiB of CPU memory, it'll not work (most likely it will get killed with cpu-oom if cgroups are configured).

Note: when it comes to running out of CPU memory regardless of which memory usage reporting tool you use - typically what you want to track is the Resident Set Size metric, which is also known as RSS (e.g., it's one of the column names in the output of `top`). There are many other metrics, but those are usually not useful for this particular need.

As I'm writing this I have this problem where I'm trying to fit a huge model into a given number of GPUs and I'm forced to offload some of the model parameters to CPU memory since I can't fit them all into GPU memory, but I'm also running out of CPU memory. So what I do is I scale down the setup to remove half the layers of the model I try to use to measure the memory footprint and then I should be able to extrapolate the required memory for the full model. Of course, the

other way is to do math, to calculate how much memory each tensor consumes, but often it's quicker to just measure usage empirically since math is often insufficient as some components get missed in the calculations. Or potentially you could get more CPU memory ;)

Observation: recently each GPU generation has been getting a sizeable increase in their memory size, however for some reason CSPs continue giving the same amount of CPU memory per compute node as they did with older GPUS with less memory, which leads to multiple problems and limitations. If you're a CSP reading this please consider future nodes to have at least the same amount of CPU memory as the total GPU memory of the node and then some - at least double or triple would be the best. Thank you!

If all you care about is the CPU peak memory report for the program you launched, you can use the `-f '%M'` flag:

```
$ /usr/bin/time -f '%M' python -c "import torch"
640684
```

Now you can, for example, feed this number to some other program - say, you want to get the peak memory usage in a human readable format:

```
$ /usr/bin/time -f '%M' python -c "import torch" |& perl -ne 'chomp; printf "%0.2fGiB\n",
$_/2**20'
0.61GiB
```

You can see that it about matched "Maximum resident set size" from before.

Note: the Unix memory measurements are often imprecise, because the memory management is very complex, so if you re-run this example again and again you will see slightly different results.

Now before we can use this as a reliable measurement tool let's check if the reported RSS memory usage checks out:

```
/usr/bin/time -f '%M' python -c "import torch"
640704
$ /usr/bin/time -f '%M' python -c "import torch; t=torch.zeros(2**14,2**14)"
1549504
$ /usr/bin/time -f '%M' python -c "import torch; t=torch.zeros(2**14,2**15)"
2588624
```

The first run is to measure the peak memory that was used to run `import torch`, which amounts to ~625MiB (`640704 / 2**10`).

Then the second run gives us the same plus memory that was needed to allocate a tensor of `2**14` by `2**14` in fp32 (default `torch dtype`) - so the expected additional memory usage is `2**14*2**14*4 = 1073741824` (fp32 `dtype` needs 4 bytes per element) or 1024MiB (`1073741824/2**20`). So let's compare the difference: `(1549504 - 640704) / 2**10` => 887.5MiB, so the reported memory came quite short of what we may have expected.

The third run is expected to have used a double of the additional memory used by the 2nd run, since we now allocated a 2x larger tensor of shape `2**14` by `2**15` - following the same math, that tensor would need 2048MiB of additional CPU memory. And the difference is `(2588624 - 640704) / 2**10` => 1902MiB so we are again short by about the same amount as the second run vs the first one.

However, if we compare the difference in the reported memory used between the second and the third run: `(2588624-1549504) / 2**10` => 1014MiB it now does check out very closely, since the expected memory difference was 1024MiB.

So what's going on here? What is being measured is the peak memory usage, so when we fire off `import torch` it allocates some memory, but also releases some, so when we add additional commands, their memory allocation will use some of the memory freed when `import torch` finished its run. So now you understand that comparing peak memory usage can be tricky if some memory get released, after being allocated.

Thus this tool is always useful to tell you how much memory was used at the highest point, but it can be tricky comparing memory usages of program variations.

Now the next question you're likely to ask is what if you have a launcher that spawns other sub-processes. Will it measure the peak memory usage of those sub-processes as well? Usually it does, but I think I have seen situations when it didn't. So let's spawn a sub-process which will run the same `import torch`:

```
$ /usr/bin/time -f '%M' sh -c 'python -c "import torch" & wait'
640744
$ /usr/bin/time -f '%M' python -c "import torch"
640836
```

We get a very similar report with and without a sub-process.

If I'm not mistaken it only follows the immediate child process, and not further, since if I use a launcher that calls another launcher which only then runs PyTorch processes I get a lot less memory reported.

cgmemtime

[cgmemtime] is a little gem of a C program that uses cgroups v2 to measure the peak CPU memory usage of a process and all of its descendants no matter how many generations follow it.

It's super easy to build:

```
git clone https://github.com/gsauthof/cgmemtime
cd cgmemtime
make
```

Now copy the binary to some folder that in your `$PATH` (hint: run `echo $PATH` to see the options) and you can start using it.

```
$ cgmemtime python -c "import torch"

user:   0.856 s
sys:    0.101 s
wall:   0.989 s
child_RSS_high:     389572 KiB
group_mem_high:     209240 KiB

$ cgmemtime sh -c 'python -c "import torch" & wait'

user:   0.875 s
sys:    0.085 s
wall:   0.961 s
child_RSS_high:     389416 KiB
group_mem_high:     206520 KiB
```

As you can see it reports both the time and the peak memory usage.

Let's compare with `/usr/bin/time`:

```
$ /usr/bin/time -f '%M' sh -c 'python -c "import torch" & wait'
389748
```

It's almost identical `389416` vs `389748` - Linux CPU memory reporting is a very fluid thing and you're very likely to get slightly different reporting running the same command.

Note: Always recalibrate your tools before making comparisons. You will see different numbers in different sections of the book for the same commands since it's likely they were run at different times with different versions on different systems.

As of 2026-08 most Unix systems have moved to cgroups v2, but it's possible to still find some older distributions that use cgroups v1. If that's the case look at older versions of `cgmemtime` since originally it was written for cgroups v1.

request: I'm yet to figure out how to make it work on a k8s pod, probably has something to do with the container not being configured properly to allow custom cgroups groups. If you know what needs to be done please share the solution.

Debugging Tensors

When developing software or dealing with some bugs during training or inference, or writing unit tests we often need to investigate tensors - their data, their attributes or both. In the following sections we are going to dive into the more efficient ways of doing this work.

Many ways to dump tensor's values

Sometimes it's just enough to print the contents of the tensor to do some visual comparisons. What gets dumped can be controlled via `torch.set_printoptions`. Here are the most useful config options with annotation:

```
torch.set_printoptions(
    threshold=100000000, # print all data (without ... skipping) - can be huge!
    sci_mode=False,      # print all data on the same scale of 1 (this disables scientific
notation)
    precision=6,         # print X decimal points for floats (default 4)
    edgeitems=5,         # when the data is large and skipped, control how many entries are
printed on each edge
    linewidth=120,       # redefine linewidth for when lines are \n-wrapped in printout (default
80)
                         # if threshold is defined, matrix printing will ignore this setting
    profile="full",      # printing defaults: "default", "short", "full"
)
```

```
$ python -c "import torch; t = torch.rand(100,100); print(t)"
tensor([[0.5171, 0.5535, 0.4281,  ..., 0.3363, 0.4250, 0.4631],
        [0.0597, 0.0126, 0.8424,  ..., 0.2475, 0.6926, 0.1892],
        [0.3671, 0.1032, 0.5224,  ..., 0.5822, 0.1384, 0.2008],
        ...,
        [0.1887, 0.9825, 0.8571,  ..., 0.9336, 0.5340, 0.6141],
        [0.0550, 0.9550, 0.4814,  ..., 0.7614, 0.0469, 0.7668],
```

```
        [0.3372, 0.4856, 0.9879,  ..., 0.8719, 0.7916, 0.1137]])
```

I often find that when the tensor values are wildly different, forcing the scientific format helps with comparing 2 tensors:

```
$ python -c "import torch; t = torch.rand(100,100); torch.set_printoptions(sci_mode=True);
print(t)"
tensor([[5.7340e-01, 6.1205e-02, 5.5568e-01,  ..., 9.7872e-01, 6.3079e-01, 1.4958e-01],
        [6.5187e-01, 7.1725e-01, 7.4311e-01,  ..., 1.6829e-01, 2.9124e-01, 9.6725e-01],
        [2.0276e-01, 7.1093e-01, 1.5570e-01,  ..., 8.5468e-01, 3.3631e-02, 7.2699e-01],
        ...,
        [1.3556e-01, 4.1345e-02, 1.1752e-01,  ..., 5.0029e-01, 9.4572e-01, 1.4204e-01],
        [8.9816e-01, 1.4840e-01, 7.5320e-01,  ..., 2.6070e-01, 8.3193e-01, 9.8864e-01],
        [2.9861e-01, 8.4406e-01, 6.4992e-01,  ..., 2.2556e-01, 7.4448e-01, 1.7672e-01]])
```

Sometimes the default 4 decimal places isn't enough, so we can ask for 6 with `precision=6`:

```
$ python -c "import torch; t = torch.rand(100,100); torch.set_printoptions(precision=6);
print(t)"
tensor([[0.496257, 0.768222, 0.088477,  ..., 0.604651, 0.109958, 0.212090],
        [0.970375, 0.836909, 0.281987,  ..., 0.670873, 0.202043, 0.489091],
        [0.521034, 0.822312, 0.122040,  ..., 0.127788, 0.704833, 0.331873],
        ...,
        [0.871615, 0.080840, 0.672732,  ..., 0.029196, 0.967139, 0.003688],
        [0.296027, 0.953120, 0.260675,  ..., 0.031883, 0.182623, 0.509600],
        [0.273938, 0.079908, 0.413711,  ..., 0.252290, 0.399835, 0.980202]])
```

As you can see each entry now shows 6 digits after the decimal instead of the default 4 - useful when two tensors look identical at 4 places but differ further out.

In all the examples so far most entries were removed and only the first and the last 3 rows and columns were dumped. But sometimes when the tensor is small we might want to see more data, so let's get 4 entries on each edge:

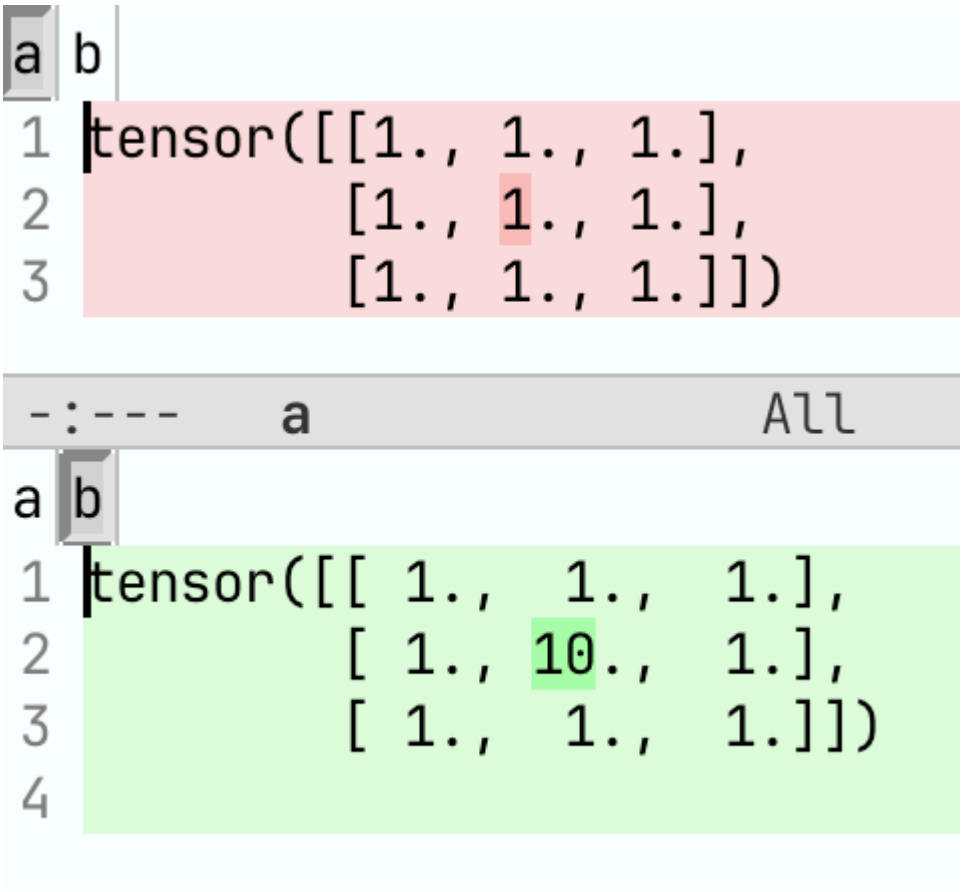
```
$ python -c "import torch; t = torch.rand(100,100); torch.set_printoptions(edgeitems=4);
print(t)"
tensor([[0.0840, 0.9232, 0.3730, 0.9597,  ..., 0.9191, 0.0434, 0.2139, 0.5933],
        [0.9864, 0.9947, 0.9185, 0.4594,  ..., 0.3290, 0.4087, 0.8190, 0.9482],
        [0.5856, 0.2450, 0.8197, 0.0203,  ..., 0.1945, 0.5485, 0.1075, 0.8870],
        [0.9267, 0.1619, 0.2912, 0.3130,  ..., 0.2847, 0.0935, 0.7931, 0.5177],
        ...,
        [0.0474, 0.9387, 0.7414, 0.3986,  ..., 0.8736, 0.9317, 0.3980, 0.3655],
        [0.8092, 0.1236, 0.3780, 0.8210,  ..., 0.8251, 0.1988, 0.8153, 0.1905],
        [0.7281, 0.0439, 0.4908, 0.4739,  ..., 0.7540, 0.4446, 0.8081, 0.0948],
        [0.0794, 0.0217, 0.4084, 0.8729,  ..., 0.9080, 0.2556, 0.8687, 0.2528]])
```

`threshold` allows you to print more data than the default, so for example if you're seeking a needle in a haystack, where most data points are identical but only a few rows or elements are off, you could dump the whole tensor into a file and then run `diff -u a b` between 2 dumps. To exemplify, let's pick a very small 3x3 tensor of 1s and then insert 10 as the

needle into the 2nd tensor in position `[1,1]` (the middle):

```
$ python -c "import torch; t = torch.ones(3,3); torch.set_printoptions(threshold=1e10); print(t)"
> a
$ python -c "import torch; t = torch.ones(3,3); t[1,1]=10;
torch.set_printoptions(threshold=1e10); print(t)" > b
$ diff -u a b
--- a   2025-11-12 02:46:18.000000000 +0000
+++ b   2025-11-12 02:46:25.000000000 +0000
@@ -1,3 +1,3 @@
-tensor([[1., 1., 1.],
-        [1., 1., 1.],
-        [1., 1., 1.]])
+tensor([[ 1.,  1.,  1.],
+        [ 1., 10.,  1.],
+        [ 1.,  1.,  1.]])
```

We can see the needle now. Of course, you'd use that in much larger tensors and will probably want to run the diff in some good visual editor so it's much easier to visualize the differences. Here is an example of comparing `a` and `b` in Emacs:



The differences are high-lighted and are easy to see, especially when the real tensors are float numbers with many decimals.

Granted, you don't need to set `set_printoptions(threshold=1e10)` for a 3x3 tensor, so try the above with 100x100. If you don't `set_printoptions(threshold=1e10)` and the needle entry ends up in what `torch` hides in `...` you will not find it.

You can accomplish something similar with `set_printoptions(profile="full")` as explained in the following paragraph.

For convenience, you also have the profiles that you can set via `profile` argument - for example, to get the full tensor set: `set_printoptions(profile="full")`. The 3 types of profile as of 2026-08 are:

- "default": what you normally get with 3 entries on each edge of the tensor, 4 decimal places for floats.
- "short": 2 entries and 2 decimal places for floats.
- "full": print all elements using scientific notation.

Let's demo the "short" profile:

```
$ python -c "import torch; t = torch.rand(100,100); torch.set_printoptions(profile='short');
print(t)"
tensor([[0.01, 0.29,  ..., 0.02, 0.41],
        [0.55, 0.34,  ..., 0.42, 0.36],
        ...,
        [0.33, 0.93,  ..., 0.76, 0.25],
        [0.21, 0.22,  ..., 0.37, 0.43]])
```

Visual debuggers like VSCode or PyCharm are excellent at showing tensor's contents and are much easier to navigate and understand than `pdb`, where you have to manually control the visualization. I would often step through to some breakpoint copy-n-paste the contents of a tensor before and after into 2 files and then run a comparison between the 2 to see the differences.

To dump not just the tensor values and possibly a few default attributes (like `device`, and `dtype`), but all of its attributes you can use `rich.inspect`

```
$ python -c "import torch, rich; t = torch.rand(2,3); rich.inspect(t)"
╭──────────────────────────────────── <class 'torch.Tensor'> ─────────────────────────────────────╮
│ ╭─────────────────────────────────────────────────────────────────────────────────────────────╮ │
│ │ tensor([[0.2276, 0.8454, 0.6496],                                                           │ │
│ │ │   │   [0.9643, 0.5291, 0.8428]])                                                          │ │
│ ╰─────────────────────────────────────────────────────────────────────────────────────────────╯ │
│                                                                                               │
│          data = tensor([[0.2276, 0.8454, 0.6496],                                             │
│                         [0.9643, 0.5291, 0.8428]])                                            │
│        device = device(type='cpu')                                                            │
│         dtype = torch.float32                                                                 │
│          grad = None                                                                          │
│       grad_fn = None                                                                          │
│             H = tensor([[0.2276, 0.9643],                                                     │
│                         [0.8454, 0.5291],                                                     │
│                         [0.6496, 0.8428]])                                                    │
│          imag = RuntimeError('imag is not implemented for tensors with non-complex dtypes.') │
│        is_cpu = True                                                                          │
│       is_cuda = False                                                                         │
│        is_ipu = False                                                                         │
│       is_leaf = True                                                                          │
│       is_maia = False                                                                         │
│       is_meta = False                                                                         │
│     is_mkldnn = False                                                                         │
```

```
|        is_mps = False                                                                                                |
|       is_mtia = False                                                                                                |
|     is_nested = False                                                                                                |
|  is_quantized = False                                                                                                |
|     is_sparse = False                                                                                                |
| is_sparse_csr = False                                                                                                |
|     is_vulkan = False                                                                                                |
|        is_xla = False                                                                                                |
|        is_xpu = False                                                                                                |
|      itemsize = 4                                                                                                    |
|        layout = torch.strided                                                                                        |
|            mH = tensor([[0.2276, 0.9643],                                                                            |
|                         [0.8454, 0.5291],                                                                            |
|                         [0.6496, 0.8428]])                                                                           |
|            mT = tensor([[0.2276, 0.9643],                                                                            |
|                         [0.8454, 0.5291],                                                                            |
|                         [0.6496, 0.8428]])                                                                           |
|          name = None                                                                                                 |
|         names = (None, None)                                                                                         |
|        nbytes = 24                                                                                                   |
|          ndim = 2                                                                                                    |
|     output_nr = 0                                                                                                    |
|          real = tensor([[0.2276, 0.8454, 0.6496],                                                                    |
|                         [0.9643, 0.5291, 0.8428]])                                                                   |
| requires_grad = False                                                                                                |
|  retains_grad = False                                                                                                |
|         shape = torch.Size([2, 3])                                                                                   |
|             T = tensor([[0.2276, 0.9643],                                                                            |
|                         [0.8454, 0.5291],                                                                            |
|                         [0.6496, 0.8428]])                                                                           |
└──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────┘
```

This allows you to quickly peek inside the tensor object. Except there might be too much information.

Getting tensor attributes

shape

Probably the most useful property attribute during debug is `t.shape` since the most common error one is likely to see is:

```
$ python -c "import torch; x=torch.ones(2,3); y=x@x"
[...]
RuntimeError: mat1 and mat2 shapes cannot be multiplied (2x3 and 2x3)
```

So you'd look at the shapes of the tensors using: `print(x.shape)` and correct your code to do the right thing - e.g. you need to transpose the 2nd tensor in the silly example above:

```
$ python -c "import torch; x=torch.ones(2,3); y=x@x.T"
```

and the error is gone.

To check the shape of the tensor, it's just `t.shape`:

```
$ python -c "import torch; x=torch.ones(2,3); print(x.shape)"
torch.Size([2, 3])
```

It's also super handy for getting various model dimensions, you're often likely to see code like this:

```
batch_size, seqlen, hidden_size = hidden_states.shape
batch_size = hidden_states.shape[0]
```

`t.shape` is the same as `t.size()`, except the latter is a callable and `t.size()[1]` is not as intuitive as `t.shape[1]`

device

Forgetting to move a tensor or a model to the right device results in the second most common error:

```
$ python -c "import torch; x=torch.ones(2,3); y=x.T.clone().cuda(); x@y"
[...]
RuntimeError: Expected all tensors to be on the same device, but got mat2 is on
cuda:0, different from other tensors on cpu
```

Moving both devices to save device - `cuda` in this case solves the problem:

```
$ python -c "import torch; x=torch.ones(2,3); y=x.T.clone().cuda(); x=x.cuda(); x@y"
```

It's a good practice to create the device you want to work with ahead of time and then create all your tensors on that device. For example, to create a `device` variable on the first gpu and then to directly allocate a tensor on it:

```
import torch
device = torch.device(f"cuda:0")
x = torch.ones(2,3, device=device)
```

To check the device of the tensor, it's just `t.device`:

```
$ python -c "import torch; x=torch.ones(2,3); print(x.device)"
cpu
```

There are various devices depending on the hardware you use, `cpu` is always there, `meta` is a special device that requires no hardware and which uses no storage, but only stores the tensor metadata, `cuda` is for gpus, and there are many others.

if there are multiple devices of the same type, they can be indexed in via the `:index` - for example, `cuda:2` - typically means the 3rd gpu (though `CUDA_VISIBLE_DEVICES` environment variable can alter the physical order of GPUs).

Probably the most common line in ML code you have seen is:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

which automatically picks the first gpu if it's available otherwise falls back to cpu.

dtype

The precision of the tensor is define via its `dtype` and the typical error involving it is:

```
$ python -c "import torch; x=torch.ones(2, dtype=torch.float16); \
y=torch.ones(2, dtype=torch.bfloat16); x@y"
[...]
RuntimeError: dot : expected both vectors to have same dtype, but found Half and BFloat16
```

Here we have the first fp16 tensor can't be `matmul`'ed by second bf16 tensor. However, there is no problem with normal multiplication:

```
$ python -c "import torch; x=torch.ones(2, dtype=torch.float16); \
y=torch.ones(2, dtype=torch.bfloat16); z=x*y; print(z.dtype)"
torch.float32
```

I didn't actually know what the dtype of the outcome will be - surprisingly the outcome of fp16 multiplied by bf16 is fp32! And now you know how to discover the `dtype` of the tensor.

To correct the first problem we need to have both tensors of the same `dtype`:

```
$ python -c "import torch; x=torch.ones(2, dtype=torch.float16); \
y=torch.ones(2, dtype=torch.float16); x@y"
```

No error this time.

Dumping multiple tensor attributes

When doing complex debugging often we need to dump multiple tensor attributes. Here the Python's self-declaring `f"{x=}` notation helps to make the code most maintainable/readable:

```
$ python -c "import torch; x=torch.ones(2, dtype=torch.float16); \
print(f'x: {x.dtype=} {x.device=} {x.shape}')"
x: x.dtype=torch.float16 x.device=device(type='cpu') torch.Size([2])
```

If you do this a lot and want a whole lot of tensor attributes using a helper debug utility can avoid a lot of typing. For example, these are the most common attributes that you might need while debugging PyTorch applications:

```
def get_tensor_metadata(t, tensor_name=None, formatted=False):
    """ dump useful tensor attributes
        Args:
        - `t`: tensor
        - `tensor_name`: tensor name (Optional)
        - `formatted`: whether to pretty format (by default just a space separated string)
        Returns: string with passed tensor attributes
    """
    # a mix of callable and non-callable tensor attributes in the defined order
    attrs = """
        device
        dtype
        shape
        numel
        requires_grad
        grad
        stride
        is_contiguous
        data_ptr
    """.split()

    def get_attr_may_be_callable(t, attr):
        a = getattr(t, attr)
        return a() if callable(a) else a

    txt = [f"{a}={get_attr_may_be_callable(t, a)}" for a in attrs]
    if tensor_name is not None:
        txt.insert(0, f"{tensor_name}:")
    if formatted:
        return "\n".join(txt)
    else:
        return ", ".join(txt)
```

Please note that some of these attributes are properties, while others are callables, therefore you'd use `t.shape`, but `t.numel()`.

Now you can call it and see the many details about the tensor:

```
x = torch.ones(2, dtype=torch.float16)
print(get_tensor_metadata(x, "x", formatted=True))
x:
device=cpu             # device type
dtype=torch.float16    # precision
shape=torch.Size([2])  # shape
numel=2                # how many elements in tensor
requires_grad=False    # does it require gradients?
grad=None              # the .grad field
stride=(1,)            # the layout of data in the storage
is_contiguous=True     # is the storage layout contiguous
data_ptr=483278528     # a unique data pointer in the memory
```

I annotated each attribute in the output if you're not familiar with some of those attributes.

If you do a lot of comparisons, a one liner format would be easier to work with - we just set `formatted=False` or remove it altogether as it's the default value:

```
print(get_tensor_metadata(x, "x"))
```

which prints:

```
x:, device=cpu, dtype=torch.float16, shape=torch.Size([2]), numel=2,
requires_grad=False, grad=None, stride=(1,),
is_contiguous=True, data_ptr=483278528
```

note: I edited the output to break the long liner into multiple lines to fit the width here.

Until recently Python couldn't tell us the name of the variable that was passed to a function, hence we need to pass the variable name as a string. Since Python 3.14 there is a way to overcome this, using template strings (t-strings). Unlike an f-string, which immediately evaluates to a `str` and throws away where its values came from, a t-string (written `t"..."`) evaluates to a `string.templatelib.Template` object that preserves both the *value* of each interpolation and the original *source text* of the expression that produced it. That source text is exactly the variable name we were missing:

```
def dbg(template):
    for item in template.interpolations:
        print(f"{item.expression} = {item.value!r}")

x = 5
data = [1, 2, 3]

dbg(t"{x}")
dbg(t"{data}")
dbg(t"{x + 1}")
```

which prints:

```
x = 5
data = [1, 2, 3]
x + 1 = 6
```

Each `Interpolation` carries `.expression` (the source text, e.g. `"x"`) and `.value` (the evaluated object), so a debug helper can report a variable's name automatically, without you having to pass it as a separate string, but you have to wrap it in `t{}`, so it's not automatic and you have to change all the callers.

Applying this to our helper, here is a variant of `get_tensor_metadata` that drops the `tensor_name` argument and recovers the name straight from the t-string:

```
def get_tensor_metadata_py314(template, formatted=False):
```

```
    """ like get_tensor_metadata, but recovers the tensor name from a t-string

        e.g. instead of `get_tensor_metadata(x, "x")` you now call
`get_tensor_metadata_py314(t"{x}")`
    """
    interpolation = template.interpolations[0]
    return get_tensor_metadata(interpolation.value, interpolation.expression,
formatted=formatted)
```

Note that you do have to wrap the tensor in a t-string at the call site - passing it plainly as `get_tensor_metadata_py314(x)` would not work, because a normal argument arrives as just a value, with no record of the `x` expression that produced it. What the t-string buys you is not having to repeat the name as a separate string - `t"{x}"` carries both the value and the name `"x"`, which looks more like a hack but it does work.

```
x = torch.ones(2, dtype=torch.float16)
print(get_tensor_metadata_py314(t"{x}", formatted=True))
```

which prints:

```
x:
device=cpu
dtype=torch.float16
[...] # output truncated for brevity
```

Sometimes having too much data dumped can make the debugging process slower, so it's up to you how many/which attributes you want to dump while debugging.

In some very complex situations you might want to dump all possible attributes. There is a secret private util that will do it for you:

```
$ python -c "import torch; x=torch.ones(2, dtype=torch.float16); \
print(torch._subclasses.meta_utils.MetaTensorDescriber().describe_tensor(x))"
MetaTensorDesc(id=0, ndim=1, dtype=torch.float16, device=device(type='cpu'),
size=torch.Size([2]), dynamo_dynamic_indices=[], layout=torch.strided,
is_inference=False, is_leaf=True, requires_grad=False, is_sparse=False,
is_mkldnn=False, is_functorch_wrapped=False, is_batchedtensor=False,
is_legacy_batchedtensor=False, is_gradtrackingtensor=False, is_view=False,
is_nested=False, nested_int=None, is_traceable_wrapper_subclass=False,
is_functional=False, is_conj=False, is_neg=False, is_parameter=False, stride=(1,),
storage_offset=0, storage=MetaStorageDesc(id=0, size=4, data=None), sparse_dim=None,
dense_dim=None, is_coalesced=None, crow_indices=None, col_indices=None,
ccol_indices=None, row_indices=None, values=None, unwrapped=None, bdim=None,
base=None, attrs=None, creation_meta=None, grad=None, ctx=None, type=None,
fake_mode=None, view_func=_CustomViewFunc(func=<built-in method _view_func_unsafe of
Tensor object at 0x100c30230>), level=None, current_level=None, functorch_stack=None,
autograd_meta_from=None, data=None)
```

That's a lot of attributes! But it's missing attributes like `t.data_ptr` and probably others.

Remember `torch.set_printoptions` from earlier? Another secret private attribute dumping approach is this context manager:

```
with torch._tensor_str.printoptions(threshold=0, edgeitems=0): print(x)
```

prints: `tensor([...], size=(2,), dtype=torch.float16)`, since we told it to hide the payload, and it ends up only dumping the most common `shape/size` and `dtype` attributes.

Now you know multiple ways of aiding your debug process introspecting tensor attributes.

Auto-dumping desired tensor attributes

Say, you have been using `print` to dump tensor's contents at strategic places in the code.

```
print(t)
```

Now you have run into the most common issue of PyTorch complaining about mismatching tensor shapes in some operation. Well, you could go and change the tensor dumps to either print the shape: `print(t.shape)` or add an additional print:

```
print(t)
print(t.shape)
```

but that's a lot of work.

When you do:

```
t = torch.rand((2,3))
print(t)
```

behind the scenes, `torch.Tensor.__repr__` is called - which is a special method python calls on an object if it's available before printing a custom representation of the object. This prints:

```
tensor([[0.6220, 0.7673, 0.9156],
        [0.8413, 0.4410, 0.9822]])
```

Why not change `__repr__` to do what we want instead. On top of python module we can add:

```
import torch
torch.Tensor.__repr_orig__ = torch.Tensor.__repr__
torch.Tensor.__repr__ = lambda t: f"{t.shape} {torch.Tensor.__repr_orig__(t)}"
```

Now this code:

```
t = torch.rand((2,3))
print(t)
```

prints both the shape and the contents of the tensor:

```
torch.Size([2, 3]) tensor([[0.6220, 0.7673, 0.9156],
        [0.8413, 0.4410, 0.9822]])
```

If all your care about is the shape, then this will do the trick:

```
import torch
torch.Tensor.__repr_orig__ = torch.Tensor.__repr__
torch.Tensor.__repr__ = lambda t: f"{t.shape}"
```

Now this code:

```
t = torch.rand((2,3))
print(t)
```

prints just the tensor's shape:

```
torch.Size([2, 3])
```

You can tweak it further to dump other tensor attributes of your choice.

lovely-tensors

There is also the lovely-tensors library that is designed to automate the process of making the tensor dump more useful and provide shapes, statistics, nan/inf counts, tensor memory usage, etc.

```
pip install lovely-tensors
```

```
import lovely_tensors as lt
lt.monkey_patch()
t = torch.rand((2,3))
print(t)
```

prints:

```
tensor[2, 3] n=6 x∈[0.295, 0.897] μ=0.512 σ=0.229 [[0.651, 0.897, 0.295], [0.446, 0.310, 0.472]]
```

It prints the shape, number of elements, various statistics and a short snippet of values. In this example the tensor is very small so it printed out all values.

It flags `inf`s and `nan`s:

```
import lovely_tensors as lt
lt.monkey_patch()
t = torch.rand((1024,1024))
t[100,100] = float('inf')
t[100,101] = float('nan')
print(t)
```

prints:

```
tensor[1024, 1024] n=1048576 (4MiB) x∈[2.980e-07 |████████▉| 1.000] μ=0.500 σ=0.288 +Inf! NaN!
```

So the output contains `+Inf! NaN!`. Also here the tensor is big enough and it prints the size of the tensor `4MiB`.

If you want the original tensor dump in addition to the other attributes, use `t.v`:

```
print(t.v)
```

gives:

```
tensor[1024, 1024] n=1048576 (4MiB) x∈[2.980e-07 |████████▉| 1.000] μ=0.500 σ=0.288 +Inf! NaN!
tensor([[0.8077, 0.8768, 0.7949,  ..., 0.4144, 0.2792, 0.6595],
        [0.1443, 0.7482, 0.0300,  ..., 0.6591, 0.1642, 0.2081],
        [0.2470, 0.1187, 0.4105,  ..., 0.3548, 0.8849, 0.3677],
        ...,
        [0.8501, 0.3632, 0.5207,  ..., 0.7146, 0.2554, 0.6134],
        [0.3375, 0.6647, 0.5794,  ..., 0.2235, 0.3606, 0.7333],
        [0.3291, 0.9476, 0.3935,  ..., 0.9475, 0.6077, 0.5302]])
```

If the tensor has `t.grad is not None` it'll also print useful stats about the grad in the output:

```
import lovely_tensors as lt
lt.monkey_patch()
t = torch.rand((2,2), requires_grad=True)
print(t)
y = t.sum().backward()
print(t)
```

so we get grad stats as well:

```
tensor[2, 2] n=4 x∈[0.184, 0.831] μ=0.500 σ=0.353 grad=None [[0.779, 0.184], [0.207, 0.831]]
tensor[2, 2] n=4 x∈[0.184, 0.831] μ=0.500 σ=0.353 grad={ x∈[1.000, 1.000] μ=1.000 σ=0. } [[0.779,
0.184], [0.207, 0.831]]
```

It has a lot of functionality for working with image tensors as well.

Detecting problematic tensor values

See also Numerical instabilities in the training chapter, which covers training-level causes and remedies for `inf`/`nan` values.

Inf

Infinity in the context of Machine Learning typically happens where as a result of a computation one or more elements of the tensor overflow.

Let's use fp16 floating point representation for demonstrating how we end up with Infinity numbers. `65504` is the largest normal floating point number that can be represented in the fp16 precision. This is slightly below `2**16` due to how this 16 bit number is represented. For details see this.

Thus we can observe:

```
$ python -c "import torch; print(torch.tensor(65504, dtype=torch.float16))"
tensor(65504., dtype=torch.float16)
$ python -c "import torch; print(torch.tensor(65504, dtype=torch.float16) + 50)"
tensor(inf, dtype=torch.float16)
```

The first tensor is fine, but the last one overflows when I added `50` to it and we get `inf`. If you remember back in the day, models were trained in fp16 mixed precision regime and this `inf` happened a lot, thus a special scaler was used to move the numbers into the safe numerical range. And that's the reason why bf16 superseded fp16, since while being less precise bf16's dynamic range is almost as big as that of fp32 despite it having only 16 bits vs. 32 bits for fp32.

To create an `inf` value on demand:

```
$ python -c "import torch; print(torch.tensor(float('inf')))"
tensor(inf)
```

To check whether a tensor contains `inf` values:

```
torch.isinf(t).any() # at least one Inf
torch.isinf(t).all() # all values are Inf
```

I created a special tool for helping to detect Overflow and Underflow values layer by layer, which can be found at Underflow and Overflow Detection.

NaN

`NaN` stands for not-a-number - you're most likely to see this in the loss during model training, typically this happens when the learning rate is too high, or the data is really bad, the optimizer fails to do its work and the loss literally breaks becoming a `NaN`.

In the previous section we explained that when a floating point number overflows it becomes an `inf`. `inf` and `nan` are very related, because `inf` turns into `nan` quite easily, e.g. multiplying `0` by `inf`:

```
$ python -c "import torch; print(0*torch.tensor(float('inf')))"
tensor(nan)
```

Most of the time `nan` happens to one or more gradient values during `backward` pass, and once `loss` becomes a NaN it's impossible to recover from it.

To check whether a tensor contains `nan` values:

```
torch.isnan(t).any() # at least one NaN
torch.isnan(t).all() # all values are NaN
```

So to debug one would need to find which layer and model parameters hit `nan` gradients. But in some situation it's the loss function that fails. Here is an example:

```
from transformers import AutoModelForCausalLM
model = AutoModelForCausalLM.from_pretrained("gpt2")
loss = model.loss_function(
    logits=torch.rand(3, 100),
    labels=torch.tensor([-100, -100, -100]),
    vocab_size=100,
)
```

As of `transformers==4.57.1` the above will give you `loss=tensor(nan)`. The issue here is that the special `-100` label masks tokens to be excluded from the loss calculation and in the above example, we have `0` tokens that aren't masked, since all labels are `-100`. And unfortunately the loss function fails and returns a NaN, instead of `0` - this is most likely a bug in the loss function implementation which makes an assumption that a sample has at least one unmasked token. But if you do sequence sharding and you use SFT you may have huge parts of the sample masked out and you can easily end up with a sample shard where all tokens are masked out. I have run into this problem when developing Arctic Long Sequence Training. The original solution I used was:

```
if all((shift_labels == -100).squeeze()):
    loss = (logits.sum() * 0.0).float()
```

Here we prevent `loss=NaN` situation and instead create an artificial loss `0`, which will also set all the grads to `0` in `backward` - the effect of this is akin to a perfect score where the model needs no adjustment since grads will be all zeros.

You can see it in context here. Though the code has evolved since then, and you can find a more elaborate version here in the loss calculation across sequence parallel ranks section.

Underflow and Overflow Detection

For this section we are going to use the underflow_overflow library.

If you start getting `loss=NaN` or the model inhibits some other abnormal behavior due to `inf` or `nan` in activations or weights one needs to discover where the first underflow or overflow happens and what led to it. Luckily you can accomplish that easily by activating a special module that will do the detection automatically.

Let's use a `t5-large` model for this demonstration.

```
from .underflow_overflow import DebugUnderflowOverflow
from transformers import AutoModel

model = AutoModel.from_pretrained("t5-large")
debug_overflow = DebugUnderflowOverflow(model)
```

[`underflow_overflow.DebugUnderflowOverflow`] inserts hooks into the model that immediately after each forward call will test input and output variables and also the corresponding module's weights. As soon as `inf` or `nan` is detected in at least one element of the activations or weights, the program will assert and print a report like this (this was caught with `google/mt5-small` under fp16 mixed precision):

```
Detected inf/nan during batch_number=0
Last 21 forward frames:
abs min  abs max  metadata
                  encoder.block.1.layer.1.DenseReluDense.dropout Dropout
0.00e+00 2.57e+02 input[0]
0.00e+00 2.85e+02 output
[...]
                  encoder.block.2.layer.0 T5LayerSelfAttention
6.78e-04 3.15e+03 input[0]
2.65e-04 3.42e+03 output[0]
             None output[1]
2.25e-01 1.00e+04 output[2]
                  encoder.block.2.layer.1.layer_norm T5LayerNorm
8.69e-02 4.18e-01 weight
2.65e-04 3.42e+03 input[0]
1.79e-06 4.65e+00 output
                  encoder.block.2.layer.1.DenseReluDense.wi_0 Linear
2.17e-07 4.50e+00 weight
1.79e-06 4.65e+00 input[0]
2.68e-06 3.70e+01 output
                  encoder.block.2.layer.1.DenseReluDense.wi_1 Linear
8.08e-07 2.66e+01 weight
1.79e-06 4.65e+00 input[0]
1.27e-04 2.37e+02 output
                  encoder.block.2.layer.1.DenseReluDense.dropout Dropout
0.00e+00 8.76e+03 input[0]
0.00e+00 9.74e+03 output
                  encoder.block.2.layer.1.DenseReluDense.wo Linear
1.01e-06 6.44e+00 weight
```

```
0.00e+00 9.74e+03 input[0]
3.18e-04 6.27e+04 output
                  encoder.block.2.layer.1.DenseReluDense T5DenseGatedGeluDense
1.79e-06 4.65e+00 input[0]
3.18e-04 6.27e+04 output
                  encoder.block.2.layer.1.dropout Dropout
3.18e-04 6.27e+04 input[0]
0.00e+00      inf output
```

The example output has been trimmed in the middle for brevity.

The second column shows the value of the absolute largest element, so if you have a closer look at the last few frames, the inputs and outputs were in the range of `1e4`. So when this training was done under fp16 mixed precision the very last step overflowed (since under `fp16` the largest number before `inf` is `64e3`). To avoid overflows under `fp16` the activations must remain way below `1e4`, because `1e4 * 1e4 = 1e8` so any matrix multiplication with large activations is going to lead to a numerical overflow condition.

At the very start of the trace you can discover at which batch number the problem occurred (here `Detected inf/nan during batch_number=0` means the problem occurred on the first batch).

Each reported frame starts by declaring the fully qualified entry for the corresponding module this frame is reporting for. For example, consider this frame:

```
                  encoder.block.2.layer.1.layer_norm T5LayerNorm
8.69e-02 4.18e-01 weight
2.65e-04 3.42e+03 input[0]
1.79e-06 4.65e+00 output
```

Here, `encoder.block.2.layer.1.layer_norm` indicates that it was a layer norm in `layer.1` of `block.2` of the encoder (both are 0-indexed, i.e. the 2nd sub-layer of the 3rd block). And the specific calls of the `forward` is `T5LayerNorm`.

Let's look at the last few frames of that report:

```
Detected inf/nan during batch_number=0
Last 21 forward frames:
abs min  abs max  metadata
[...]
                  encoder.block.2.layer.1.DenseReluDense.wi_0 Linear
2.17e-07 4.50e+00 weight
1.79e-06 4.65e+00 input[0]
2.68e-06 3.70e+01 output
                  encoder.block.2.layer.1.DenseReluDense.wi_1 Linear
8.08e-07 2.66e+01 weight
1.79e-06 4.65e+00 input[0]
1.27e-04 2.37e+02 output
                  encoder.block.2.layer.1.DenseReluDense.wo Linear
1.01e-06 6.44e+00 weight
0.00e+00 9.74e+03 input[0]
3.18e-04 6.27e+04 output
                  encoder.block.2.layer.1.DenseReluDense T5DenseGatedGeluDense
```

```
1.79e-06 4.65e+00 input[0]
3.18e-04 6.27e+04 output
                  encoder.block.2.layer.1.dropout Dropout
3.18e-04 6.27e+04 input[0]
0.00e+00      inf output
```

The last frame reports for `Dropout.forward` function with the first entry for the only input and the second for the only output. You can see that it was called from an attribute `dropout` inside `DenseReluDense` class. We can see that it happened in `layer.1` of `block.2` (the 2nd sub-layer of the 3rd block), during the very first batch. Finally, the absolute largest input elements was `6.27e+04` and same for the output was `inf`.

You can see here, that `T5DenseGatedGeluDense.forward` resulted in output activations, whose absolute max value was around 62.7K, which is very close to fp16's top limit of 64K. In the next frame we have `Dropout` which renormalizes the weights, after it zeroed some of the elements, which pushes the absolute max value to more than 64K, and we get an overflow (`inf`).

As you can see it's the previous frames that we need to look into when the numbers start going into very large for fp16 numbers.

Let's match the report to the code from [`models/t5/modeling_t5.py`]:

```
class T5DenseGatedGeluDense(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.wi_0 = nn.Linear(config.d_model, config.d_ff, bias=False)
        self.wi_1 = nn.Linear(config.d_model, config.d_ff, bias=False)
        self.wo = nn.Linear(config.d_ff, config.d_model, bias=False)
        self.dropout = nn.Dropout(config.dropout_rate)
        self.gelu_act = ACT2FN["gelu_new"]

    def forward(self, hidden_states):
        hidden_gelu = self.gelu_act(self.wi_0(hidden_states))
        hidden_linear = self.wi_1(hidden_states)
        hidden_states = hidden_gelu * hidden_linear
        hidden_states = self.dropout(hidden_states)
        hidden_states = self.wo(hidden_states)
        return hidden_states
```

Now it's easy to see the `dropout` call, and all the previous calls as well.

Since the detection is happening in a forward hook, these reports are printed immediately after each `forward` returns.

Going back to the full report, to act on it and to fix the problem, we need to go a few frames up where the numbers started to go up and most likely switch to the `fp32` mode here, so that the numbers don't overflow when multiplied or summed up. Of course, there might be other solutions. For example, we could turn off `amp` temporarily if it's enabled, after moving the original `forward` into a helper wrapper, like so:

```
import torch

def _forward(self, hidden_states):
    hidden_gelu = self.gelu_act(self.wi_0(hidden_states))
```

```
    hidden_linear = self.wi_1(hidden_states)
    hidden_states = hidden_gelu * hidden_linear
    hidden_states = self.dropout(hidden_states)
    hidden_states = self.wo(hidden_states)
    return hidden_states

def forward(self, hidden_states):
    if torch.is_autocast_enabled():
        with torch.amp.autocast("cuda", enabled=False):
            return self._forward(hidden_states)
    else:
        return self._forward(hidden_states)
```

Since the automatic detector only reports on inputs and outputs of full frames, once you know where to look, you may want to analyze the intermediary stages of any specific `forward` function as well. In such a case you can use the `detect_overflow` helper function to inject the detector where you want it, for example:

```
from underflow_overflow import detect_overflow


class T5LayerFF(nn.Module):
    [...]

    def forward(self, hidden_states):
        forwarded_states = self.layer_norm(hidden_states)
        detect_overflow(forwarded_states, "after layer_norm")
        forwarded_states = self.DenseReluDense(forwarded_states)
        detect_overflow(forwarded_states, "after DenseReluDense")
        return hidden_states + self.dropout(forwarded_states)
```

You can see that we added 2 of these and now we track if `inf` or `nan` for `forwarded_states` was detected somewhere in between.

Actually, the detector already reports these because each of the calls in the example above is a `nn.Module`, but let's say if you had some local direct calculations this is how you'd do that.

Additionally, if you're instantiating the debugger in your own code, you can adjust the number of frames printed from its default, e.g.:

```
from .underflow_overflow import DebugUnderflowOverflow

debug_overflow = DebugUnderflowOverflow(model, max_frames_to_save=100)
```

Specific batch absolute min and max value tracing

The same debugging class can be used for per-batch tracing with the underflow/overflow detection feature turned off.

Let's say you want to watch the absolute min and max values for all the ingredients of each `forward` call of a given batch, and only do that for batches 1 and 3. Then you instantiate this class as:

```
debug_overflow = DebugUnderflowOverflow(model, trace_batch_nums=[1, 3])
```

And now full batches 1 and 3 will be traced using the same format as the underflow/overflow detector does.

Batches are 0-indexed.

This is helpful if you know that the program starts misbehaving after a certain batch number, so you can fast-forward right to that area. Here is a sample truncated output for such configuration:

```
                  *** Starting batch number=1 ***
abs min  abs max  metadata
                  shared Embedding
1.01e-06 7.92e+02 weight
0.00e+00 2.47e+04 input[0]
5.36e-05 7.92e+02 output
[...]
                  decoder.dropout Dropout
1.60e-07 2.27e+01 input[0]
0.00e+00 2.52e+01 output
                  decoder T5Stack
     not a tensor output
                  lm_head Linear
1.01e-06 7.92e+02 weight
0.00e+00 1.11e+00 input[0]
6.06e-02 8.39e+01 output
                   T5ForConditionalGeneration
     not a tensor output

                  *** Starting batch number=3 ***
abs min  abs max  metadata
                  shared Embedding
1.01e-06 7.92e+02 weight
0.00e+00 2.78e+04 input[0]
5.36e-05 7.92e+02 output
[...]
```

Here you will get a huge number of frames dumped - as many as there were forward calls in your model, so it may or may not be what you want, but sometimes it can be easier to use for debugging purposes than a normal debugger. For example, if a problem starts happening at batch number 150. So you can dump traces for batches 149 and 150 and compare where numbers started to diverge.

You can also specify the batch number after which to stop the training, with:

```
debug_overflow = DebugUnderflowOverflow(model, trace_batch_nums=[1, 3], abort_after_batch_num=3)
```

Floating point math discrepancies on different devices

See also Reproducibility for achieving determinism across different software and hardware setups.

It's important to understand that depending on which device the floating point math is performed on the outcomes can

be different. For example doing the same floating point operation on a CPU and a GPU may lead to different outcomes, similarly when using 2 different GPU architectures, and even more so if these are 2 different types of accelerators (e.g. NVIDIA vs. AMD GPUs).

Here is an example of discrepancies I was able to get doing the same simple floating point math on an 11 Gen Intel i7 CPU and an NVIDIA A100 80GB (PCIe) GPU:

```
import torch

def do_math(device):
    inv_freq = (10 ** (torch.arange(0, 10, device=device) / 10))
    print(f"{inv_freq[9]:.20f}")
    return inv_freq.cpu()

a = do_math(torch.device("cpu"))
b = do_math(torch.device("cuda"))

torch.testing.assert_close(a, b, rtol=0.0, atol=0.0)
```

when we run it we get 2 out of 10 elements mismatch:

```
7.94328212738037109375
7.94328308105468750000
[...]
AssertionError: Tensor-likes are not equal!

Mismatched elements: 2 / 10 (20.0%)
Greatest absolute difference: 9.5367431640625e-07 at index (9,)
Greatest relative difference: 1.200604771156577e-07 at index (9,)
```

This was a simple low-dimensional example, but in reality the tensors are much bigger and will typically end up having more mismatches.

Now you might say that the `1e-6` discrepancy can be safely ignored. And it's often so as long as this is a final result. If this tensor from the example above is now fed through a 100 layers of `matmul`s, this tiny discrepancy is going to compound and spread out to impact many other elements with the final outcome being quite different from the same action performed on another type of device.

For example, see this discussion - the users reported that when doing Llama-2-7b inference they were getting quite different logits depending on how the model was initialized. To clarify the initial discussion was about DeepSpeed potentially being the problem, but in later comments you can see that it was reduced to just which device the model's buffers were initialized on. The trained weights aren't an issue they are loaded from the checkpoint, but the buffers are recreated from scratch when the model is loaded, so that's where the problem emerges.

It's uncommon that small variations make much of a difference, but sometimes the difference can be clearly seen, as in this example where the same image is produced on a CPU and an MPS device.

To better illustrate the problem that motivated me to post this issue, I can present some output from CURL. This first image is from the CPU:



This is from the mps device:



This snapshot and the commentary come from this PyTorch Issue thread.

If you're curious where I pulled this code from - this is a simplified reduction of this original code in modeling_llama.py:

```
class LlamaRotaryEmbedding(nn.Module):
    def __init__(self, dim, max_position_embeddings=2048, base=10_000, device=None):
        super().__init__()

        self.dim = dim
        self.max_position_embeddings = max_position_embeddings
        self.base = base
        inv_freq = 1.0 / (self.base ** (torch.arange(0, self.dim, 2).float().to(device) /
self.dim))
        self.register_buffer("inv_freq", inv_freq, persistent=False)
```

Diagnosing crashes, hangs and tracing execution

If you suspect a crash is caused by faulty hardware rather than your code, see also Troubleshooting NVIDIA GPUs.

Dealing with Async CUDA bugs

When using CUDA, failing pytorch programs very often produce a python traceback that makes no sense or can't be acted upon. This is because due to CUDA's async nature - when a CUDA kernel is executed, the program has already moved on and when the error happened the context of the program isn't there. The async functionality is there to make things faster, so that while the GPU is churning some `matmul` the program on CPU could already start doing something else.

At other times some parts of the system will actually tell you that they couldn't generate the correct traceback, as in this error:

```
[E ProcessGroupNCCL.cpp:414] Some NCCL operations have failed or timed out. Due to the
asynchronous nature of CUDA kernels, subsequent GPU operations might run on corrupted/
incomplete data. To avoid this inconsistency, we are taking the entire process down.
```

There are a few solutions.

If the failure is instant and can be reproduced on CPU (not all programs work on CPU), simply re-rerun it after hiding your GPUs. This is how you do it:

```
CUDA_VISIBLE_DEVICES="" python my-pytorch-program.py
```

The env var `CUDA_VISIBLE_DEVICES` is used to manually limit the visibility of GPUs to the executed program. So for example if you have 8 gpus and you want to run program1.py with first 4 gpus and program2.py with the remaining 2 gpus you can do:

```
CUDA_VISIBLE_DEVICES="0,1,2,3" python my-pytorch-program1.py
CUDA_VISIBLE_DEVICES="4,5,6,7" python my-pytorch-program2.py
```

and the second program won't be the wiser that it's not using GPUs 0-3.

But in the case of debug we are hiding all GPUs, by setting `CUDA_VISIBLE_DEVICES=""`.

Now the program runs on CPU and you will get a really nice traceback and will fix the problem in no time.

But, of course, if you your program requires multiple GPUs this won't work. And so here is another solution.

Rerun your program after setting this environment variable:

```
CUDA_LAUNCH_BLOCKING=1 python my-pytorch-program.py
```

This variable tells pytorch (or any other CUDA-based program) to turn its async nature off everywhere and now all operations will be synchronous. So when the program crashes you should now get a perfect traceback and you will know exactly what ails your program.

In theory enabling this variable should make everything run really slow, but in reality it really depends on your software. We did the whole of BLOOM-176B training using `CUDA_LAUNCH_BLOCKING=1` with `Megatron-DeepSpeed` and had zero slowdown - we had to use it as pytorch was hanging without it and we had no time to figure the hanging out.

So, yes, when you switch from async to sync nature, often it can hide some subtle race conditions, so there are times that a hanging disappears as in the example I shared above. So measure your throughput with and without this flag and sometimes it might actual not only help with getting an in-context traceback but actually solve your problem altogether.

Note: NCCL==2.14.3 coming with `pytorch==1.13` hangs when `CUDA_LAUNCH_BLOCKING=1` is used. So don't use it with that version of pytorch. The issue has been fixed in `nccl>=2.17` which should be included in `pytorch==2.0`.

Debugging CUDA kernel memory errors with `compute-sanitizer`

`CUDA_VISIBLE_DEVICES=""` and `CUDA_LAUNCH_BLOCKING=1` from the previous section give you an in-context Python traceback that points at *which* operator failed - but they can't see *inside* a CUDA kernel. When the bug is a bad memory access, a race, or use of uninitialized device memory, NVIDIA's `compute-sanitizer` (ships with the CUDA toolkit; it replaced the old `cuda-memcheck`) instruments the kernel and reports the offending access.

It has four sub-tools, selected with `--tool`:

- `memcheck` (the default) - out-of-bounds and misaligned device memory accesses, and, with `--leak-check full`, device memory that was never freed.
- `racecheck` - shared-memory data races between threads of a block.
- `initcheck` - reads of uninitialized device global memory.
- `synccheck` - invalid `__syncthreads()` / barrier use.

It runs on any process that launches CUDA kernels, `python` included - just prefix your normal command:

```
compute-sanitizer --tool memcheck python my-program.py
```

Every kernel is instrumented, so expect a large slowdown; narrow to a single reproducing step and use `--launch-count`/`--launch-skip` to check only the suspect launches.

Leak reporting is off unless you ask for it, and its absence is indistinguishable from a clean run - without `--leak-check full` a program that allocates a megabyte and never frees it still finishes with `ERROR SUMMARY: 0 errors`. Turning it on comes with a caveat on PyTorch: the caching allocator is still holding its segments when the process exits, and each one is reported as a leak. A correct script that allocates a single small tensor reports `LEAK SUMMARY: 2097156 bytes leaked in 2 allocations` - a 2MiB allocator segment plus a 4-byte block. Dropping your references and calling

`torch.cuda.empty_cache()` before exit releases the segment and leaves only the 4 bytes, which is a quick way to tell your own leak from the allocator's bookkeeping.

First, when you *don't* need it: modern PyTorch's own indexing / gather / scatter / take kernels bounds-check on the device and `assert`, so a stray index aborts with a clear message, e.g.

```
.../IndexKernel.cu:339: ... Assertion `idx < numel ... index out of bounds` failed.
```

For those, `CUDA_LAUNCH_BLOCKING=1` plus the assert already name the op and (with blocking) your exact Python line - no sanitizer needed (the wording and PyTorch source location vary by version). Reach for `compute-sanitizer` when the kernel *doesn't* self-check: your own or a third-party extension that silently corrupts memory and only crashes later somewhere unrelated.

We'll use one tiny buggy kernel, built two ways. It writes with 8 threads into a 4-int buffer, so threads 4-7 run off the end (kernel_oob.cu):

```
#include <torch/extension.h>
#include <cuda_runtime.h>

__global__ void write_oob(int *p)
{
    int i = threadIdx.x;
    p[i] = i;   // 8 threads, 4-int buffer -> threads 4-7 run off the end
}

void run()
{
    int *d;
    cudaMalloc(&d, 4 * sizeof(int));   // tight 16-byte allocation
    write_oob<<<1, 8>>>(d);
    cudaDeviceSynchronize();
    cudaFree(d);
}

PYBIND11_MODULE(TORCH_EXTENSION_NAME, m)
{
    m.def("run", &run, "launch the buggy kernel");
}
```

When you don't have the kernel source

The usual case: the kernel is inside a prebuilt wheel - a release build with no line info, and you don't have the `.cu`. To see exactly what that looks like, build our kernel the way wheels ship it - *without* `-lineinfo` (kernel_oob_prebuilt.py):

```
#!/usr/bin/env python
import torch
from torch.utils.cpp_extension import load

# built the way a PyPI wheel ships: release, NO -lineinfo / -G
```

```
ext = load(
    name="kernel_oob_prebuilt",
    sources=["kernel_oob.cu"],
    verbose=False,
)
ext.run()
torch.cuda.synchronize()
```

```
compute-sanitizer --tool memcheck python kernel_oob_prebuilt.py
```

memcheck still catches the bad write and pins it to the kernel and to your Python call site - but with no source line inside the kernel:

```
========= Invalid __global__ write of size 4 bytes
=========     at write_oob(int *)+0x50
=========     by thread (4,0,0) in block (0,0,0)
=========     Access to 0x7f...10 is out of bounds
=========     and is 1 bytes after the nearest allocation at 0x7f...00 of size 16 bytes
=========     Saved host backtrace up to driver entry point at kernel launch time
=========         Host Frame: run() in kernel_oob_prebuilt.so
=========         [ ... pybind frames ... ]
=========         Host Frame: <module> in kernel_oob_prebuilt.py:11
========= ERROR SUMMARY: 8 errors
```

You get the kernel (`write_oob(int *)+0x50`), the exact overrun (1 byte past a 16-byte allocation), and your Python launch site (`kernel_oob_prebuilt.py:11`) - just not a line *inside* the kernel, which is impossible without a build you don't have. For a real third-party op that's usually enough: which kernel, which of your tensors, which call - then fix the inputs or file a bug upstream.

One PyTorch-specific gotcha: if the overrun is on a normal `torch` tensor, the caching allocator hands out sub-regions of a big `cudaMalloc` slab, so a small overrun stays *inside* the slab and memcheck sees nothing. Disable caching so it can:

```
PYTORCH_NO_CUDA_MEMORY_CACHING=1 compute-sanitizer --tool memcheck python my-program.py
```

(Our demo allocates its buffer with a raw `cudaMalloc`, so it fires without this - you'll need it for OOBs on real tensors.)

When you have the kernel source

If the kernel is *yours*, you control the build, so add `-lineinfo` and memcheck will additionally name the exact offending line (kernel_oob.py):

```
#!/usr/bin/env python
import torch
from torch.utils.cpp_extension import load

ext = load(
    name="kernel_oob",
```

```
    sources=["kernel_oob.cu"],
    extra_cuda_cflags=["-lineinfo"],
    verbose=True,
)
ext.run()
torch.cuda.synchronize()
```

```
compute-sanitizer --tool memcheck python kernel_oob.py
```

Same report, now with the source line:

```
========= Invalid __global__ write of size 4 bytes
=========     at write_oob(int *)+0x50 in kernel_oob.cu:7
=========     Access to 0x7f...10 is out of bounds
=========     and is 1 bytes after the nearest allocation at 0x7f...00 of size 16 bytes
=========         Host Frame: run() in kernel_oob.so
=========         [ ... pybind frames ... ]
=========         Host Frame: <module> in kernel_oob.py:11
========= ERROR SUMMARY: 8 errors
```

The only difference from the prebuilt run is `in kernel_oob.cu:7` - the exact store. Use `-lineinfo` (keeps optimizations) or `-G` for a full debug build (disables optimizations, much slower). Triton kernels work too and emit line info by default.

Shared-memory races with `racecheck`

memcheck asks whether a kernel touched memory it shouldn't. `racecheck` asks whether two threads of a block touched the same `__shared__` location with nothing ordering them - the bug behind a kernel that returns slightly different numbers on every run instead of crashing. It sees shared memory only, so it is a tool for hand-written kernels; a program that just calls PyTorch ops has no shared memory of its own to check.

kernel_race.cu makes the classic mistake - each thread publishes a value and reads its neighbor's with no `__syncthreads()` in between (built with `-lineinfo` by kernel_race.py, so the report can name lines):

```
__global__ void race(int *out)
{
    __shared__ int s[64];
    int i = threadIdx.x;
    s[i] = i;                    // written by thread i ...
    out[i] = s[(i + 1) % 64];    // ... read by thread i-1, no barrier between
}
```

```
compute-sanitizer --tool racecheck python kernel_race.py
```

```
========= Error: Race reported between Write access at race(int *)+0xc0 in kernel_race.cu:8
=========     and Read access at race(int *)+0xf0 in kernel_race.cu:9 [256 hazards]
========= RACECHECK SUMMARY: 1 hazard displayed (1 error, 0 warnings)
```

That is the default `analysis` mode - one record per racing pair, with the hazard count behind it. `--racecheck-report hazard` prints those 256 hazards individually instead, each naming the two threads, the shared address and the value seen, plus a host backtrace - so pair it with `--print-limit` to cap the output and `--print-level error` to keep only the hazards crossing a warp boundary (8 of the 256 here); the rest are warnings tagged `(Warp Level Programming)`, since warp-synchronous code sometimes relies on them deliberately. To re-read findings without re-running an instrumented program, `--save race.log` records them and `--read race.log` prints them back; the file name takes `%p` for the pid, so `torchrun` ranks don't overwrite each other.

Primary reference: NVIDIA Compute Sanitizer.

Stepping inside a kernel with `cuda-gdb`

`compute-sanitizer` tells you that a kernel did something wrong and where. `cuda-gdb`, from the same toolkit, lets you stop inside it and look around - it is `gdb` with device support: breakpoints in `__global__` functions, per-thread inspection, and reads of device memory.

Build the kernel with `-G` instead of `-lineinfo` - full device debug info with optimizations off (kernel_oob_gdb.py) - and put your normal command after `--args`. The wait on the first run is the compile rather than the debugger: this example took 102s the first time and 17s on later runs, once the built extension was cached. One PyTorch-specific wrinkle: `cpp_extension.load` compiles the kernel while the program is already running, so the symbol doesn't exist when the debugger starts and the breakpoint has to be allowed to stay pending:

```
$ cuda-gdb -ex 'set breakpoint pending on' -ex 'break write_oob' -ex run --args python
kernel_oob_gdb.py
Function "write_oob" not defined.
Breakpoint 1 (write_oob) pending.
[Switching focus to CUDA kernel 0, grid 1, block (0,0,0), thread (0,0,0), device 0, sm 124, warp
0, lane 0]

CUDA thread hit Breakpoint 1.1, write_oob<<<(1,1,1),(8,1,1)>>> (p=0x7ffc6fe00000) at
kernel_oob.cu:6
6           int i = threadIdx.x;
```

From there the usual `gdb` vocabulary works, with device additions: `print p[0]` reads device memory, `info args` shows the kernel's arguments, and `cuda thread (4,0,0)` moves the focus to a specific thread - which is what you want in this kernel, where threads 0-3 are fine and only 4-7 run off the end.

Reference: NVIDIA cuda-gdb.

segfaults and getting a backtrace from a core file

It's not uncommon for a complex pytorch program to segfault and drop a core file. Especially if you're using complex extensions like NCCL.

The corefile is what the program generates when it crashes on a low-level - e.g. when using a python extension - such as a CUDA kernel or really any library that is coded directly in some variant of C or another language and made accessible in python through some binding API. The most common cause of a segfault is when such software accesses memory it has not allocated. For example, a program may try to free memory it hasn't allocated. But there could be many other reasons.

When a segfault event happens Python can't do anything, as the proverbial carpet is pulled out from under its feet, so it can't generate an exception or even write anything to the output.

In these situation one must go and analyze the libC-level calls that lead to the segfault, which is luckily saved in the core

file.

The general mechanics - enabling core dumps (`ulimit -c unlimited` and `kernel.core_pattern`), loading a core file into `gdb`, and the `bt` / `bt full` / `thread apply all bt` commands - are covered in Segmentation fault, core files and gdb. Here we focus on the PyTorch-specific nuances.

If your program crashed, you will often find a file that will look something like: `core-python-3097667-6`

A PyTorch-specific gotcha: you must load the core file with the exact python executable that ran the crashed program. If you have multiple python environments you have to activate the right one first, or `gdb` may fail to unpack the core file. So typically I'd go:

```
conda activate my-env
gdb python core-python-3097667-6
```

- adjust `my-env` to whatever env you use, or instead of conda use whatever way you use to activate your python environment - and perhaps you're using the system-wise python and then you don't need to activate anything.
- adjust the name of the core file to the file you have gotten - it's possible that there are many - pick the latest then.

Now `gdb` will churn for a bit and will give you a prompt where you type: `bt`. We will use an actual core file here:

```
(gdb) bt
#0  0x0000147539887a9f in raise () from /lib64/libc.so.6
#1  0x000014753985ae05 in abort () from /lib64/libc.so.6
#2  0x000014751b85a09b in __gnu_cxx::__verbose_terminate_handler() [clone .cold.1] () from
/lib64/libstdc++.so.6
#3  0x000014751b86053c in __cxxabiv1::__terminate(void (*)()) () from /lib64/libstdc++.so.6
#4  0x000014751b860597 in std::terminate() () from /lib64/libstdc++.so.6
#5  0x000014751b86052e in std::rethrow_exception(std::__exception_ptr::exception_ptr) () from
/lib64/libstdc++.so.6
#6  0x000014750bb007ef in c10d::ProcessGroupNCCL::WorkNCCL::handleNCCLGuard() ()
   from .../python3.8/site-packages/torch/lib/libtorch_cuda_cpp.so
#7  0x000014750bb04c69 in c10d::ProcessGroupNCCL::workCleanupLoop() ()
   from.../python3.8/site-packages/torch/lib/libtorch_cuda_cpp.so
#8  0x000014751b88cba3 in execute_native_thread_routine () from /lib64/libstdc++.so.6
#9  0x000014753a3901cf in start_thread () from /lib64/libpthread.so.0
#10 0x0000147539872dd3 in clone () from /lib64/libc.so.6
```

and there you go. How do you make sense of it?

Well, you go from the bottom of the stack to the top. You can tell that a `clone` call was made in `libc` which then called `start_thread` in `libpthread` and then if you keep going there are a bunch of calls in the torch libraries and finally we can see that the program terminated itself, completing with `raise` from `libc` which told the Linux kernel to kill the program and create the core file.

This wasn't an easy to understand backtrace.

footnote: Yes, python calls it a *traceback* and elsewhere it's called a *backtrace* - it's confusing, but it's more or less the same thing.

Actually I had to ask pytorch devs for help and received:

- PyTorch `ProcessGroup` watchdog thread caught an asynchronous error from NCCL
- This error is an `"unhandled system error"` which in this particular case turned out to be an IB-OPA error
- The `ProcessGroup`'s `WorkCleanUp` thread rethrew the error so that the main process would crash and the user would get notified (otherwise this async error would not surface)

Trust me there are times when even if you're inexperienced the backtrace can give you enough of a hint to where you should look for troubleshooting.

But fear not - most of the time you won't need to understand the traceback. Ideally you'd just attach the core file to your filed Issue. But it can easily be 5GB large. So the developers that will be trying to help you will ask you to generate a `gdb` backtrace and now you know how to do that.

Since most modern programs run multiple threads and `bt` only shows the main thread, use `thread apply all bt` at the `(gdb)` prompt to get a backtrace for every thread (see the link above for details).

I didn't promise it'll be easy, I just showed you where to start.

strace

Similar to py-spy, `strace` is a super-useful tool which traces any running application at the low-level system calls - e.g. `libC` and alike.

For example, run:

```
strace python -c "print('strace')"
```

and you will see everything that is done at the system call level as the above program runs.

But usually it's more useful when you have a stuck program that spins all CPU cores at 100% but nothing happens and you want to see what's it doing. In this situation you simply attached to the running program like so:

```
strace --pid PID
```

where you get the PID for example from the output of `top` or `ps`. Typically I just copy-n-paste the PID of the program that consumes the most CPU - `top` usually shows it at the very top of its listing.

Same as `py-spy` you may need `sudo` perms to attached to an already running process - it all depends on your system setup. But you can always start a program with `strace` as I have shown in the original example.

Let's look at a small sub-snippet of the output of `strace python -c "print('strace')"`

```
write(1, "strace\n", 7strace
)                = 7
```

Here we can see that a write call was executed on filedescriptor `1`, which almost always is `stdout` (`stdin` being `0`, and `stderr` being `2`).

If you're not sure what a filedescriptor is pointing to, normally you can tell from `strace`'s output itself. But you can also do:

```
ls -l /proc/PID/fd
```

where PID is the pid of the currently running program you're trying to investigate.

For example, when I run the above while running a pytest test with gpus, I got (partial output):

```
l-wx------ 1 stas stas 64 Mar  1 17:22 5 -> /dev/null
lr-x------ 1 stas stas 64 Mar  1 17:22 6 -> /dev/urandom
lrwx------ 1 stas stas 64 Mar  1 17:22 7 -> /dev/nvidiactl
lrwx------ 1 stas stas 64 Mar  1 17:22 8 -> /dev/nvidia0
lr-x------ 1 stas stas 64 Mar  1 17:22 9 -> /dev/nvidia-caps/nvidia-cap2
```

so you can see that a device `/dev/null` is open as FD (file descriptor) `5`, `/dev/urandom` as FD 6, etc.

`lsof` gives you the same mapping without going through `/proc`, and adds the file type and the access mode:

```
$ lsof -p PID
COMMAND     PID USER   FD   TYPE DEVICE SIZE/OFF        NODE NAME
python  1876037 stas  cwd    DIR  9,127      143 22548711123 /tmp/enospc
python  1876037 stas  rtd    DIR  0,285     4096 12884906136 /
python  1876037 stas  txt    REG  0,285 30598912 19864359926 /home/stas/envs/dev/bin/python3.12
python  1876037 stas    0r   CHR    1,3      0t0           6 /dev/null
python  1876037 stas    1w  FIFO   0,15      0t0  2216140539 pipe
python  1876037 stas    2w  FIFO   0,15      0t0  2216140540 pipe
python  1876037 stas    3w   REG  9,127        1 22548711151 /tmp/enospc/holdme.txt
python  1876037 stas    4r   CHR    1,9      0t0          11 /dev/urandom
...
```

The `FD` column carries the mode - `3w` is FD 3 open for writing, `4r` is FD 4 open for reading - and the special entries `cwd`, `rtd` and `txt` are the working directory, the root directory and the executable itself. Add `-n` to skip hostname lookups and `-P` to skip port-name lookups, both of which make it noticeably faster on a busy machine.

It also answers the question `/proc` can't - *who* has this file open:

```
$ lsof /tmp/enospc/holdme.txt
COMMAND     PID USER   FD   TYPE DEVICE SIZE/OFF        NODE NAME
python  1876037 stas    3w   REG  9,127        1 22548711151 /tmp/enospc/holdme.txt
```

which is how you find the process still holding a deleted-but-not-freed file, or the one keeping a mount busy.

Now let's go look at another snippet from our `strace` run.

```
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
```

Here it tried to see if file `/etc/ld.so.preload` exists, but as we can see it doesn't - this can be useful if some shared library is missing - you can see where it's trying to load it from.

Let's try another one:

```
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libpthread.so.0", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\0\0\0\0\0\0\0"..., 832) = 832
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=21448, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 16424, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f8028807000
mmap(0x7f8028808000, 4096, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1000) =
0x7f8028808000
mmap(0x7f8028809000, 4096, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) =
0x7f8028809000
mmap(0x7f802880a000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000)
= 0x7f802880a000
close(3)
```

here we can see that it opens `/lib/x86_64-linux-gnu/libpthread.so.0` and assigns it FD 3, it then reads 832 chars from FD 3, (we can also see that the first chars are ELF - which stands for a shared library format), then memory maps it and closes that file.

In this following example, we see a python cached file is opened, its filepointer is moved to 0, and then it's read and closed.

```
openat(AT_FDCWD, "/home/stas/anaconda3/envs/py38-pt113/lib/
python3.8/__pycache__/abc.cpython-38.pyc", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0664, st_size=5329, ...}) = 0
lseek(3, 0, SEEK_CUR)                   = 0
lseek(3, 0, SEEK_CUR)                   = 0
fstat(3, {st_mode=S_IFREG|0664, st_size=5329, ...}) = 0
brk(0x23bf000)                          = 0x23bf000
read(3, "U\r\r\n\0\0\0\0\24\216\177c\211\21\0\0\343\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 5330) =
5329
read(3, "", 1)                          = 0
close(3)
```

It's important to notice that file descriptors are re-used, so we have seen the same FD 3 twice, but each time it was open to a different file.

If your program is for example trying to reach to the Internet, you can also tell these calls from `strace` as the program would be reading from a socket file descriptor.

So let's run an example on a program that downloads files from the HF hub:

```
strace python -c 'import sys; from transformers import AutoConfig;
AutoConfig.from_pretrained(sys.argv[1])' t5-small
```

here is some relevant to this discussion snippet:

```
socket(AF_INET6, SOCK_STREAM|SOCK_CLOEXEC, IPPROTO_TCP) = 3
setsockopt(3, SOL_TCP, TCP_NODELAY, [1], 4) = 0
ioctl(3, FIONBIO, [1])                  = 0
connect(3, {sa_family=AF_INET6, sin6_port=htons(443), sin6_flowinfo=htonl(0), inet_pton(AF_INET6,
"2600:1f18:147f:e850:e203:c458:10cd:fc3c
```

```
", &sin6_addr), sin6_scope_id=0}, 28) = -1 EINPROGRESS (Operation now in progress)
poll([{fd=3, events=POLLOUT|POLLERR}], 1, 10000) = 1 ([{fd=3, revents=POLLOUT}])
getsockopt(3, SOL_SOCKET, SO_ERROR, [0], [4]) = 0
[...]
write(3, "\26\3\3\0F\20\0\0BA\4\373m\244\16\354/\334\205\361j\225\356\202m*\305\332\275\251\
17J"..., 126) = 126
read(3, 0x2f05c13, 5)                   = -1 EAGAIN (Resource temporarily unavailable)
poll([{fd=3, events=POLLIN}], 1, 9903)  = 1 ([{fd=3, revents=POLLIN}])
read(3, "\24\3\3\0\1", 5)               = 5
read(3, "\1", 1)                        = 1
read(3, "\26\3\3\0(", 5)                = 5
read(3, "\0\0\0\0\0\0\0\344\v\273\225`\4\24m\234~\371\332%l\364\254\34\3472<\0356s\313"..., 40)
= 40
ioctl(3, FIONBIO, [1])                  = 0
poll([{fd=3, events=POLLOUT}], 1, 10000) = 1 ([{fd=3, revents=POLLOUT}])
write(3, "\27\3\3\1.\0\374$\361\217\337\377\264g\215\364\345\256\260\211$\326pkR\345\276,\321\
221`-"..., 307) = 307
ioctl(3, FIONBIO, [1])                  = 0
read(3, 0x2ef7283, 5)                   = -1 EAGAIN (Resource temporarily unavailable)
poll([{fd=3, events=POLLIN}], 1, 10000) = 1 ([{fd=3, revents=POLLIN}])
```

You can see where that again it uses FD 3 but this time it opens a INET6 socket instead of a file. You can see that it then connects to that socket, polls, reads and writes from it.

A failing syscall also answers "which file?" when the exception won't. A "No space left on device" traceback names the line that wrote, but not what it was writing to:

```
Traceback (most recent call last):
  File "save.py", line 10, in <module>
    save("x" * 100000)
  File "save.py", line 8, in save
    f.write(data)
OSError: [Errno 28] No space left on device
```

The path was computed at run time, so it appears nowhere in the traceback. Re-run under `strace` with `-y`, which annotates every file descriptor with the file it points at, and filter down to the calls that matter:

```
strace -y -e trace=openat,write -o strace.txt python save.py
grep ENOSPC strace.txt
```

```
write(3</dev/full>, "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"..., 100000) = -1 ENOSPC (No space left on
device)
```

and there is the culprit. (This example writes to `/dev/full`, a device that always reports `ENOSPC`, so you can reproduce the failure without actually filling a disk.)

`-y` needs a reasonably recent `strace`. Without it you get a bare `write(3, ...)` and have to scan backwards for the `openat` that returned FD 3 - the same FD-to-path exercise as above.

If you want to reproduce a disk-full failure on demand rather than wait for one, see emulating an almost full disk partition.

There are many other super useful understandings one can derive from using this tool.

BTW, if you don't want to scroll up-down, you can also save the output to a file:

```
strace -o strace.txt python -c "print('strace')"
```

Now, since you're might want to strace the program from the very beginning, for example to sort out some race condition on a distributed filesystem, you will want to tell it to follow any forked processes. This what the `-f` flag is for:

```
strace -o log.txt -f torchrun --nproc_per_node=4 --nnodes=1 --tee 3 test.py
```

So here we launch 4 processes and will end up running `strace` on at least 5 of them - the launcher plus 4 processes (each of which may spawn further child processes).

It will conveniently prefix each line with the pid of the program so it should be easy to tell which system was made by which process.

But if you want separate logs per process, then use `-ff` instead of `-f`.

The `strace` manpage has a ton of other useful options.

While the methodologies found in this article were developed while working with multi-node multi-gpu pytorch-based training, they, of course, can help with any multi-process multi-node Python programs.

Helper tools

Try to use the following script torch-distributed-gpu-test.py to diagnose the situation.

This will help primarily with discovering network-related issues. And also to quickly understand how multi-gpu communications work.

For code-related issues read the rest of this document.

Approaches to diagnosing multi-gpu hanging / deadlocks

py-spy

First do `pip install py-spy`.

Now you can attach to each process with:

```
py-spy dump -n -p PID
```

and it will tell you where the process hangs (very often it's a nccl collective function or a `barrier`).

- `PID` is the process id of the hanging python process.
- `-n` is useful if you want to see stack traces from python extensions written in C, C++, etc., as the program may hang in one of the extensions
- you may need to add `sudo` before the command - for more details see this note.

If you have no `sudo` access your sysadmin might be able to perform this for you:

```
sudo echo 0 > /proc/sys/kernel/yama/ptrace_scope
```

which will allow you running `py-spy` (and `strace`) without needing `sudo`. Beware of the possible security implications - but typically if your compute node is inaccessible from the Internet it's less likely to be a risk.

If the above fails with "Permission denied" error, either of the following 2 ways should do the trick:

```
sudo sysctl -w kernel.yama.ptrace_scope=0
sudo bash -c "echo 0 > /proc/sys/kernel/yama/ptrace_scope"
```

To make this change permanent edit `/etc/sysctl.d/10-ptrace.conf` and set:

```
kernel.yama.ptrace_scope = 0
```

Here is an example of `py-spy dump` python stack trace:

```
Thread 835995 (active): "MainThread"
    broadcast (torch/distributed/distributed_c10d.py:1191)
    _aggregate_total_loss (deepspeed/runtime/pipe/engine.py:540)
    train_batch (deepspeed/runtime/pipe/engine.py:330)
    train_step (megatron/training.py:436)
    train (megatron/training.py:851)
    pretrain (megatron/training.py:187)
    <module> (pretrain_gpt.py:239)
```

The very first line is where the program is stuck.

If the hanging happens inside a CPP extension, add `--native` `py-spy` and it'll show the non-python code if any.

If the process has multiple threads it'll show a stack trace of each thread. For example:

```
Thread 0x7F6D3C29D740 (idle): "MainThread"
    wait (threading.py:312)
    result (concurrent/futures/_base.py:435)
    main (slurmeventd.py:208)
    <module> (slurmeventd.py:217)
Thread 0x7F6CF5FFB700 (idle): "Thread-CallbackRequestDispatcher"
    wait (threading.py:312)
    get (queue.py:171)
    _get_many (pubsub_v1/subscriber/_protocol/helper_threads.py:56)
    __call__ (pubsub_v1/subscriber/_protocol/helper_threads.py:103)
    run (threading.py:892)
    _bootstrap_inner (threading.py:954)
    _bootstrap (threading.py:912)
```

`MainThread` is the main process.

multi-process py-spy

Now, how do you do it for multiple processes. Doing it one-by-one is too slow. So let's do it at once.

If the launch command was `python`, what you do is:

```
pgrep -P $(pgrep -o python) | xargs -I {} py-spy dump --pid {}
```

if `deepspeed`:

```
pgrep -P $(pgrep -o deepspeed) | xargs -I {} py-spy dump --pid {}
```

for `accelerate`:

```
pgrep -P $(pgrep -o accelerate) | xargs -I {} py-spy dump --pid {}
```

you get the idea.

This particular approach will only analyze the main processes and not various other sub-processes/threads spawned by these processes. So if you have 8 gpus and 8 processes, the above will generate 8 stack traces.

Then you can pipe the output into this additional useful filter:

```
pgrep -P $(pgrep -o deepspeed) | xargs -I {} py-spy dump --pid {} | grep -A5 MainThread
```

so it'll show the first 5 entries of each traceback of the `MainThread`.

If you have zombies from previously run processes and they are defunct and can't be killed you most likely need to switch to `pgrep -n` to grep the newest, rather than the oldest processes (`pgrep -o`).

```
pgrep -P $(pgrep -n deepspeed) | xargs -I {} py-spy dump --pid {}
```

In some situations when an additional launcher wrapper is added that let's say calls a `deepspeed` launcher, I will see that you end up with an additional Python parent process, so then you need to add one more level of `pgrep -P`:

```
pgrep -P $(pgrep -P $(pgrep -n deepspeed)) | xargs -I {} py-spy dump --pid {}
```

If you want all processes and their subprocesses, then you'd just run:

```
pgrep -f python | xargs -I {} py-spy dump --pid {}
```

(and as before replace `python` with the name of the launcher program if it's not `python`)

multi-node py-spy via srun

What if you have multiple nodes?

You can of course `ssh` to each node interactively and dump the stack traces.

If you're using the SLURM environment you can use `srun` to do it on all nodes for you.

Now in another console get the `SLURM_JOBID` (or get it from `salloc` log):

```
squeue -u `whoami` -o "%.16i %9P %26j %.8T %.10M %.8l %.6D %.20S %R"
```

Now use the following `srun` command after adjusting jobid with `SLURM_JOBID` from the outcome of the command above this sentence:

```
srun --jobid=2180718 --gres=gpu:0 --nodes=40 --tasks-per-node=1 --output=trace-%N.out sh -c 'ps
aux | grep python | egrep -v "grep|srun" | grep `whoami` | awk "{print \$2}" | xargs -I {} py-spy
dump --native --pid {}' || echo "failed"
```

Notes:

- One must use `--gres=gpu:0` for the monitor `srun` or otherwise it will block until the main `srun` (the one running the training) exits.
- Each node will generate its unique log file named `trace-nodename.out` - so this would help to identify which node(s) are problematic. You can remove `--output=trace-%N.out` if you want it all being dumped to stdout
- In some SLURM versions you may also need to add `--overlap`
- In some SLURM versions the jobid might not match that of reported in `squeue`, so you have to get the correct `SLURM_JOB_ID` from the logs of the job you're trying to "attach" to - i.e. your `srun` job that allocated the GPUs.
- Sometimes `bash` doesn't work, but `sh` does. I think it has to do with what dot files get `sourced`
- You might need to also activate a custom python environment, which you can do like so:

```
srun --jobid=2180718 --gres=gpu:0 --nodes=40 --tasks-per-node=1 --output=trace-%N.out sh -c
'conda activate myenvname; ps auxc | ... ' || echo "failed"
```

or you can do it inside `~/.bashrc` or whatever shell's rc file you decide to use.

As mentioned before if you want just the main processes you'd use this instead:

```
srun --jobid=2180718 --gres=gpu:0 --nodes=40 --tasks-per-node=1 --output=trace-%N.out sh -c
'pgrep -P $(pgrep -o python) | xargs -I {} py-spy dump --pid {}' || echo "failed"
```

Adjust `python` if need be as explained in the multi-process py-spy section.

The previous longer command will deliver traces for all python processes.

If you're not getting anything, start with the basic debug like:

```
srun --jobid=2180718 --gres=gpu:0 --nodes=40 --tasks-per-node=1 --output=trace-%N.out sh -c
'date'
```

once you know you're talking to all the nodes, then you can progressively unravel the depth of calls, as in:

```
srun --jobid=2180718 --gres=gpu:0 --nodes=40 --tasks-per-node=1 sh -c 'date'
srun --jobid=2180718 --gres=gpu:0 --nodes=40 --tasks-per-node=1 sh -c 'pgrep -o python'
srun --jobid=2180718 --gres=gpu:0 --nodes=40 --tasks-per-node=1 sh -c 'pgrep -P $(pgrep -o
python) '
srun --jobid=2180718 --gres=gpu:0 --nodes=40 --tasks-per-node=1 sh -c 'pgrep -P $(pgrep -o
python) | xargs -I {} py-spy dump --pid {}'
```

and at each stage check that the output makes sense - e.g. the 2nd and 3rd call you should be getting the PIDs of the processes.

multi-node py-spy via pdsh

`pdsh` seems to be a good easy tool to use to accomplish remote work on multiple nodes. Say, you're running on 2 nodes with hostnames `nodename-5` and `nodename-8`, then you can quickly test that remote execution is working by getting the `date` on all of these hosts with just:

```
$ PDSH_RCMD_TYPE=ssh pdsh -w nodename-[5,8] "date"
nodename-5: Wed Oct 25 04:32:43 UTC 2023
nodename-8: Wed Oct 25 04:32:45 UTC 2023
```

footnote: `pdsh` should be available via a normal OS package installer

Once you tested that `date` works it's time to move to `py-spy`.

To do `py-spy` on all python processes that are sub-processes, it'd be:

```
PDSH_RCMD_TYPE=ssh pdsh -w nodename-[5,8] 'pgrep -P $(pgrep -o python) | xargs -I {} py-spy dump
--pid {}'
```

but as you're likely to need to have the `~/.bashrc` run, you will need to clone it into `~/.pdshrc`, reduce that clone to what is needed to be run (e.g. modify `PATH`, `activate conda`) and then `source` it, like:

```
PDSH_RCMD_TYPE=ssh pdsh -w nodename-[5,8] 'source ~/.pdshrc; pgrep -P $(pgrep -o python) | xargs
-I {} py-spy dump --pid {}'
```

The reason you need a startup script is because usually `~/.bashrc` starts with:

```
# If not running interactively, don't do anything
case $- in
    *i*) ;;
```

```
      *) return;;
esac
```

so when you run such non-interactive workflows Bash won't process its `~/.bashrc` normally (exit early) and thus anything relying on this startup script won't work. So you can either remove the non-interactive exiting code above or fork `~/.bashrc` into a startup file that only contains what's needed for the remote command to succeed.

footnote: there is nothing special about `~/.pdshrc` - any other name would do, since you're manually `source`ing it.

And if your system isn't setup to run `py-spy` w/o `sudo` as explained a few sections up, you'd need something like this:

```
PDSH_RCMD_TYPE=ssh pdsh -w nodename-[5,8] 'sudo bash -c "source ~/.pdshrc; pgrep -P $(pgrep -o
python) | xargs -I {} py-spy dump --pid {}"'
```

Of course, you may need to edit the `pgrep` section to narrow down which processes you want to watch.

Additionally, to avoid being prompted with:

```
Are you sure you want to continue connecting (yes/no/[fingerprint])?
```

for every new node you haven't logged into yet, you can disable this check with:

```
echo "Host *" >> ~/.ssh/config
echo "  StrictHostKeyChecking no" >> ~/.ssh/config
```

Here I assume you're on an isolated cluster where replaced nodes routinely change their host keys, so you don't need to worry about security issues and thus bypassing such check is most likely OK.

multi-node py-spy via ds_ssh

This is yet another way, but please make sure to read the pdsh section first.

The following notes require `pip install deepspeed`.

In one SLURM environment I also attempted using `pdsh` via `ds_ssh`, but somehow I wasn't able to run `py-spy` remotely - the main issue was that remote `ssh` command wasn't giving the same env as when I was logged in interactively via `ssh`. But if you have `sudo` access on the compute nodes then you could do:

First prepare `hostfile`:

```
function makehostfile() {
perl -e '$slots=split /,/, $ENV{"SLURM_STEP_GPUS"};
$slots=8 if $slots==0; # workaround 8 gpu machines
@nodes = split /\n/, qx[scontrol show hostnames $ENV{"SLURM_JOB_NODELIST"}];
print map { "$_ slots=$slots\n" } @nodes'
}
makehostfile > hostfile
```

Adapt `$slots` to the number of gpus per node. You may have to adapt this script if your `scontrol` produces a different

output.

Now run the `py-spy` extraction command over all participating nodes:

```
ds_ssh -f hostfile "source ~/.pdshrc; ps aux | grep python | grep -v grep | grep `whoami` | awk
'{print \$2}' | xargs -I {} sudo py-spy dump --pid {} "
```

Notes:

- Put inside `~/.pdshrc` whatever init code that you may need to run. If you don't need any you can remove `source ~/.pdshrc;` from the command line.
- If you don't have it already `ds_ssh` is installed when you do `pip install deepspeed`.
- you might need to `export PDSH_RCMD_TYPE=ssh` if you get `rcmd: socket: Permission denied` error

Diagnosing NCCL collective hangs

When a multi-GPU run hangs, `py-spy` shows *where* each rank's Python is parked - but that's usually every rank blocked inside the same NCCL wait, which doesn't reveal what went wrong. The useful questions are *which* collective failed to match across ranks and *where in your code* it was issued. PyTorch's ProcessGroupNCCL watchdog answers both, in two escalating steps: a *desync report* that names the rank and collective that fell out of step, and a *flight recorder* that dumps the full per-rank history of recent collectives, each with the Python call stack that issued it. Start with the first; reach for the second only when you need more.

This section is about the `nccl` backend's watchdog, which is what produces both reports. Measured on PyTorch 2.14.0: if the process group uses the `nccl2` implementation instead - selected with `backend="nccl2"`, or with `TORCH_DIST_USE_NCCL2=1` which redirects plain `"nccl"` to it - none of the `TORCH_NCCL_*` and `TORCH_FR_*` variables below do anything. The same hang then reports a single `Operation timed out` line and a Python traceback pointing at the collective the rank was blocked in rather than the one that diverged, with no desync report and no dump file. Reproduce the hang under `nccl` to diagnose it.

footnote: the 2.14 release notes describe a flight recorder that records through process-group hooks rather than only under NCCL. That is not what a stock 2.14.0 wheel does for `nccl2`, which is why the advice above is to switch back rather than to expect a dump.

The desync report

For most hangs the desync report is enough. Enable it before launching the program with:

```
export TORCH_NCCL_DESYNC_DEBUG=1
export TORCH_FR_BUFFER_SIZE=2000
```

`TORCH_NCCL_DESYNC_DEBUG=1` makes the watchdog, at timeout, name which collective and which rank fell out of step, and `TORCH_FR_BUFFER_SIZE` (any non-zero value turns the flight recorder on) is what adds the source stack trace of the stuck call to that report.

Let's write a buggy script collective_mismatch.py where rank `0` issues one extra `all_reduce` that the other rank never joins, so every rank blocks until the watchdog fires. Run it with:

```
torchrun --standalone --nproc_per_node=2 collective_mismatch.py
```

after setting the environment variables above.

The repro deliberately passes a short `timeout=timedelta(seconds=8)` to `init_process_group`, which is why the watchdog

fires after 8s. If `default_pg_nccl_timeout` is not overridden the default for NCCL is 10 minutes, so the watchdog won't fire until then. When you're actively chasing a hang, lower it to seconds or a few minutes so the report lands promptly but remember to then undo it before going into production. Starting from PyTorch 2.14 you don't have to edit the `init_process_group` call to change it: `dist.set_timeout(timedelta(seconds=8))` applies to the group afterwards, and the same call stretches it back out around something slow like a checkpoint load.

On timeout the watchdog names the culprit and prints the call stack of the stuck collective:

```
[Rank 0] Watchdog caught collective operation timeout: WorkNCCL(SeqNum=2, OpType=ALLREDUCE,
NumelIn=4, NumelOut=4, Timeout(ms)=8000) ran for 8097 milliseconds before timing out.
 - [0] Timeout at collective: ALLREDUCE, #3
 - To our best knowledge, the lagging/dead/mismatched ranks that caused the desync are:
     [1] finished collective #2, but didn't join collective #3 (count from 1)
Stack trace of the failed collective:
#2 buggy from collective_mismatch.py:8
#3 main from collective_mismatch.py:15
#4 <module> from collective_mismatch.py:18
```

That already points at `collective_mismatch.py:8` - the stray `all_reduce` - and, crucially, at `main:15` that called `buggy`, so you learn *how* the bad call was reached, not just that it happened. This is usually enough to locate and fix the bug.

The flight recorder trace

If you need a more detailed picture, the flight recorder keeps the full per-rank history - enable the recorder environment variables:

```
export TORCH_FR_BUFFER_SIZE=2000
export TORCH_NCCL_DUMP_ON_TIMEOUT=1
export TORCH_FR_DUMP_TEMP_FILE=/tmp/fr
```

- `TORCH_NCCL_DUMP_ON_TIMEOUT=1` - dump the buffer to disk automatically when the watchdog fires; without it the recorder still runs but nothing is written out.
- `TORCH_FR_DUMP_TEMP_FILE=/tmp/fr` - path prefix for the dump; each rank appends its own rank number, so you get `/tmp/fr0`, `/tmp/fr1`, and so on.
- `TORCH_FR_BUFFER_SIZE` is explained in the previous section.

and now re-run:

```
torchrun --standalone --nproc_per_node=2 collective_mismatch.py
```

footnote: `TORCH_FR_BUFFER_SIZE` and `TORCH_FR_DUMP_TEMP_FILE` were named `TORCH_NCCL_TRACE_BUFFER_SIZE` and `TORCH_NCCL_DEBUG_INFO_TEMP_FILE` before PyTorch 2.9; the old names still work but warn. These `TORCH_NCCL_*`/`TORCH_FR_*` variables are PyTorch-level controls, distinct from NCCL's own `NCCL_*` variables such as `NCCL_DEBUG`.

On timeout each rank writes `/tmp/fr<rank>`. The helper script collective_mismatch_analyze.py loads the pickle file and, for each collective that did not complete, prints its call site and callers with torch's own frames stripped out:

```
#!/usr/bin/env python
```

```
import pickle, glob, os, torch

SHOW_ALL   = False   # False: only collectives that hung; True: every collective
NUM_FRAMES = 2       # user frames to show: the actual call site + who called it

TORCH_DIR = os.path.dirname(torch.__file__)      # wherever torch actually lives
def is_user(frame):                              # skip torch's own collective plumbing
    return not frame["filename"].startswith(TORCH_DIR)

d = pickle.load(open(sorted(glob.glob("/tmp/fr*"))[0], "rb"))
for e in d["entries"]:
    if SHOW_ALL or e["state"] != "completed":
        print(e["state"], e["profiling_name"])
        user = [f for f in e["frames"] if is_user(f)]   # innermost first
        for f in user[:NUM_FRAMES]:                      # call site, then its caller
            print("   ", f["name"], os.path.basename(f["filename"]) + ":" + str(f["line"]))
```

```
started nccl:all_reduce
    buggy collective_mismatch.py:8
    main collective_mismatch.py:15
scheduled nccl:all_reduce_barrier
    main collective_mismatch.py:16
    <module> collective_mismatch.py:18
```

The entry left in `started` (or `scheduled`) rather than `completed` is the hang, and its first non-torch frame is the offending call - here `collective_mismatch.py:8`. If you need you can raise `NUM_FRAMES` to a higher number. Set `SHOW_ALL = True` to print the whole timeline, including the collectives that completed before the divergence:

```
completed nccl:all_reduce
    buggy collective_mismatch.py:6
    main collective_mismatch.py:15
started nccl:all_reduce
    buggy collective_mismatch.py:8
    main collective_mismatch.py:15
scheduled nccl:all_reduce_barrier
    main collective_mismatch.py:16
    <module> collective_mismatch.py:18
```

This is usually faster than reading raw stacks because it surfaces the ordering mismatch between ranks - mismatched sizes, a different collective order, or a collective one rank skipped - that causes most NCCL hangs.

Primary reference: ProcessGroupNCCL environment variables.

Network-level hanging

The hanging could be happening at the network level. `NCCL_DEBUG=INFO` can help here.

Run the script with `NCCL_DEBUG=INFO` env var and try to study the outcome for obvious errors. It will tell you which device it's using, e.g.:

```
DeepWhite:21288:21288 [0] NCCL INFO NET/Socket : Using [0]enp67s0:192.168.50.21<0>
```

So it's using interface `enp67s0` over `192.168.50.21`

Is your `192.168.50.21` firewalled? or is it somehow a misconfigured network device?

Does it work if you use a loopback device `127.0.0.1`?

```
NCCL_DEBUG=INFO NCCL_SOCKET_IFNAME=lo torchrun --nproc_per_node 4 --nnodes 1
torch-distributed-gpu-test.py
```

if not, see what other local network devices you have via `ifconfig` - try that instead of `lo` if any.

It's currently using `enp67s0` in the above example.

Isolate problematic GPUs

You can also try to see if only some GPUs fail

For example, does it work if you use the first 2 or the last 2 gpus:

```
CUDA_VISIBLE_DEVICES=0,1 torchrun --nproc_per_node 2 --nnodes 1 torch-distributed-gpu-test.py
```

then the 2nd pair:

```
CUDA_VISIBLE_DEVICES=2,3 torchrun --nproc_per_node 2 --nnodes 1 torch-distributed-gpu-test.py
```

python `trace`

Now what happens when the training doesn't just hang, but the hanging process stops responding? e.g. this happens when there is a serious hardware issue. But what if it is recurrent and `py-spy` won't help here, since it won't be able to attach to a process that is not responding.

So next came the idea of tracing all calls like one does with `strace(1)`, I researched python calls tracing facilities and have discovered that python has a `trace` sub-system.

The following code will trace all python calls and log them to the console and into a dedicated per process log file, via a custom `Tee` module I added.

This then can help to understand where some processes stopped responding, since we will have the log of the last call and all the previous calls before it went unresponsive.

```
$ cat train.py
[...]

def main():
    # [...]
    train()

import re
```

```
class Tee:
    """
    A helper class to tee print's output into a file.
    Usage:
    sys.stdout = Tee(filename)
    """

    def __init__(self, filename):
        self.stdout = sys.stdout
        self.file = open(filename, "a")

    def __getattr__(self, attr):
        return getattr(self.stdout, attr)

    def write(self, msg):
        self.stdout.write(msg)
        self.file.write(msg)
        self.file.flush()

    def flush(self):
        self.stdout.flush()
        self.file.flush()

if __name__ == "__main__":

    import sys
    import trace
    import socket
    import os

    # enable the trace
    if 0:
        cwd = os.path.realpath('.')
        pid = os.getpid()
        hostname = socket.gethostname()
        local_rank = int(os.environ["LOCAL_RANK"])
        trace_output_file = f"{cwd}/trace-{hostname}-{local_rank}-{pid}.txt"

        # create a Trace object, telling it what to ignore, and whether to
        # do tracing or line-counting or both.
        tracer = trace.Trace(
            ignoredirs=[sys.prefix, sys.exec_prefix],
            trace=1,
            count=1,
            timing=True,
        )

        # run the new command using the given tracer
        sys.stdout = Tee(trace_output_file)
        tracer.run('main()')
    else:
```

```
        main()
```

This code doesn't require any special handing other than enabling the trace by changing `if 0` to `if 1`.

If you don't set `ignoredirs`, this will now dump all python calls. Which means expect a lot of GBs of data logged, especially if you have hundreds of GPUs.

Of course, you don't have to start tracing from `main` - if you suspect a specific are you can start tracing there instead and it'll be much faster and less data to save.

I wish I could tell `trace` which packages to follow, but alas it only supports dirs to ignore, which is much more difficult to set, and thus you end up with a lot more data than needrf. But still this is a super useful tool for debugging hanging processes.

Also, your code will now run much much slower and the more packages you trace the slower it will become.

NicerTrace

As `Trace` proved to provide very limited usability when debugging a complex multi-node multi-hour run crash, I have started on working on a better version of the `trace` python module.

You can find it here: NicerTrace

I added multiple additional flags to the constructor and made the output much more useful. You fill find a full working example in that same file, just run:

```
python NicerTrace.py
```

and you should see:

```
        NicerTrace.py:1 <module>
0:00:00 <string>:     1:         NicerTrace.py:185 main
0:00:00 NicerTrace.py:   186:     img = Image.new("RGB", (4, 4))
        PIL.Image:2896 new
0:00:00 Image.py:  2912:     _check_size(size)
        PIL.Image:2875 _check_size
0:00:00 Image.py:  2883:     if not isinstance(size, (list, tuple)):
0:00:00 Image.py:  2886:     if len(size) != 2:
0:00:00 Image.py:  2889:     if size[0] < 0 or size[1] < 0:
```

as you will see in the example I set:

```
            packages_to_include=["PIL"],
```

so it'll trace `PIL` plus anything that is not under `site-packages`. If you need to trace another package, just add it to that list.

This is a very fresh work-in-progress package, so it's evolving as we are trying to make it help us resolve a very complex crashing situation.

Working with generated trace files

When the per-node-rank trace files has been generated the following might be helpful to quickly analyze the situation:

- grep for a specific match and also print the file and line number where it was found:

```
grep -n "backward" trace*
```

- show `tail -1` of all trace files followed by the name of each file:

```
find . -name "trace*" -exec sh -c 'echo "$1: $(tail -3 "$1")"' _ {} \;
```

- or similar to the above, but print 5 last lines with the leading filename and some vertical white space for an easier reading:

```
find . -name "trace*" -exec sh -c 'echo; echo $1; echo "$(tail -5 "$1")"' _ {} \;
```

- count how many times grep matched a given pattern in each ifle and print the matched file (in this example matching the pattern `backward`):

```
find . -name "trace*" -exec sh -c 'echo "$1: $(grep "backward" $1 | wc -l)"' _ {} \;
```

good old `print`

Now once you discovered where the hanging happens to further understand why this is happening, a debugger would ideally be used, but more often than not debugging multi-process (multi-node) issues can be very difficult.

In such situations a good old `print` works. You just need to add some debug prints before the calls where things hang, things that would help understand what lead to the deadlock. For example, some `barrier` was missing and one or a few processes skipped some code and while the rest of processes are still blocking waiting for everybody to send some data (for example in NCCL collective functions like `gather` or `reduce`).

You of course, want to prefix each print with the rank of the process so that you could tell which is which. For example:

```
import torch.distributed as dist
print(f"{dist.get_rank()}: passed stage 0")
```

What you will quickly discover is that if you have multiple GPUs these prints will be badly interleaved and you will have a hard time making sense of the debug data. So let's fix this. We are going to override `print` with a custom version of the same, but which uses `flock` to ensure that only one process can write to stdout at the same time.

The helper module `printflock.py` is included here. To activate it just run this at the top of the module you're debugging:

```
from printflock import printflock as print
```

and now all your `print` calls in that module will magically be non-iterleaved. You can of course, just use `printflock` directly:

```
from printflock import printflock
import torch.distributed as dist
printflock(f"{dist.get_rank()}: passed stage 0")
```

core files

If the hanging happens inside non-python code, and `py-spy --native` isn't enough for some reason you can make the hanging program dump a core file, which is done with one of these approaches:

```
gcore <pid>
kill -ABRT <pid>
```

and then you can introspect the core file as explained here.

If you don't get the core file dumped you need to configure your system to allow so and also specify where the core files should be saved to.

To ensure the file is dumped in bash run (other shells may use a different command):

```
ulimit -c unlimited
```

To make this persistent run:

```
echo '* soft core unlimited' >> /etc/security/limits.conf
```

On some systems like Ubuntu the core files are hijacked by `apport`, check the contents of `/proc/sys/kernel/core_pattern` to see where they are sent. You can override where they are sent with:

```
sudo sysctl -w kernel.core_pattern=/tmp/core-%e.%p.%h.%t
```

Change the directory if you want to, but make sure that the user the program is running under can write to that directory. To make this change permanent edit `/etc/sysctl.conf` and add `kernel.core_pattern=/tmp/core-%e.%p.%h.%t` (or modify if it's already there).

footnote: see `man core` for all the different templates available

If on Ubuntu by default it sends core files to `apport`, which may save the core to `/var/lib/apport/coredump` or `/var/crash`. But you can change this as explained above.

A quick way to test if your setup can generate a core file is:

```
sleep 10 &
kill -SEGV $!
```

Normally `SIGSEGV` isn't recommended for a real situation of diagnosing a hanging program, because `SIGSEGV` is likely to launch a sighandler, but for this test it's good enough. (`$!` is the PID of the `sleep` we just backgrounded, so only that process

is signalled.)

Code loops

Code loops can be tricky to debug in hanging scenarios. If you have code like the following:

```
for i, d in enumerate(data):
    some_hanging_call(d)
```

it's possible that one process hangs in the first iteration, and another process in the second iteration, which makes things very confusing. But the stack trace won't give such indication, as the line numbers would be the same, even though the processes aren't in the same place code progression-wise.

In such situations unroll the loop to be:

```
d_iter = iter(data)
some_hanging_call(next(d_iter))
some_hanging_call(next(d_iter))
```

and now when you run `py-spy` the line numbers will be correct. The processes hanging in the first iteration will report the first `some_hanging_call` and those in the second iteration in the second call - as each now has its own line.

Hardware-specific issues

AMD/ROCm hangs or slow with IOMMU

See Troubleshooting AMD GPUs.

Debugging multi-node training

For diagnosing NCCL connectivity problems between GPUs and nodes (the layer below PyTorch), see also How to diagnose NCCL multi-gpu and multi-node connectivity issues.

Getting nodes to talk to each other

Once you need to use more than one node to scale your training, e.g., if you want to use DDP to train faster, you have to get the nodes to talk to each other, so that communication collectives could send data to each other. This is typically done via a comms library like NCCL. And in our DDP example, at the end of training step all GPUs have to perform an `all_reduce` call to synchronize the gradients across all ranks.

In this section we will discuss a very simple case of just 2 nodes (with 8 GPUs each) talking to each other and which can then be easily extended to as many nodes as needed. Let's say that these nodes have the IP addresses 10.0.0.1 and 10.0.0.2.

Once we have the IP addresses we then need to choose a port for communications.

In Unix there are 64k ports. The first 1k are reserved for common services so that any computer on the Internet could connect to any other computer knowing ahead of time which port to connect to. For example, port 22 is reserved for SSH. So that whenever you do `ssh example.com` in fact the program open a connection to `example.com:22`.

As there are thousands of services out there, the reserved 1k ports is not enough, and so various services could use pretty much any port. But fear not, when you get your Linux box on the cloud or an HPC, you're unlikely to have many preinstalled services that could use a high number port, so most ports should be available.

Therefore let's choose port 6000.

Now we have: `10.0.0.1:6000` and `10.0.0.2:6000` that we want to be able to communicate with each other.

The first thing to do is to open port `6000` for incoming and outgoing connections on both nodes. It might be open already or you might have to read up the instructions of your particular setup on how to open a given port.

Here are multiple ways that you could use to test whether port 6000 is already open.

```
telnet localhost:6000
nmap -p 6000 localhost
nc -zv localhost 6000
curl -v telnet://localhost:6000
```

Most of these should be available via `apt install` or whatever your package manager uses.

Let's use `nmap` in this example. If I run:

```
$ nmap -p 22 localhost
[...]
PORT   STATE SERVICE
22/tcp open  ssh
```

We can see the port is open and it tells us which protocol and service is allocated as a bonus.

Now let's run:

```
$ nmap -p 6000 localhost
[...]

PORT     STATE  SERVICE
6000/tcp closed X11
```

Here you can see port 6000 is closed.

Now that you understand how to test, you can proceed to test the `10.0.0.1:6000` and `10.0.0.2:6000`.

First ssh to the first node in terminal A and test if port 6000 is opened on the second node:

```
ssh 10.0.0.1
nmap -p 6000 10.0.0.2
```

if all is good, then in terminal B ssh to the second node and do the same check in reverse:

```
ssh 10.0.0.2
nmap -p 6000 10.0.0.1
```

If both ports are open you can now use this port. If either or both are closed you have to open these ports. Since most clouds use a proprietary solution, simply search the Internet for "open port" and the name of your cloud provider.

The next important thing to understand is that compute nodes will typically have multiple network interface cards (NICs). You discover those interfaces by running:

```
$ sudo ifconfig
```

One interface is typically used by users to connecting to nodes via ssh or for various other non-compute related services - e.g., sending an email or download some data. Often this interface is called `eth0`, with `eth` standing for Ethernet, but it can be called by other names.

Then there is the inter-node interface which can be InfiniBand, EFA, OPA, HPE Slingshot, etc. (more information). There could be one or dozens of those interfaces.

Here are some examples of `ifconfig`'s output:

```
$ sudo ifconfig
enp5s0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.0.23  netmask 255.255.255.0  broadcast 10.0.0.255
        [...]
```

I removed most of the output showing only some of the info. Here the key information is the IP address that is listed after `inet`. In the example above it's `10.0.0.23`. This is the IP address of interface `enp5s0`.

If there is another node, it'll probably be `10.0.0.24` or `10.0.0.21` or something of sorts - the last segment will be the one with a different number.

Let's look at another example:

```
$ sudo ifconfig
ib0     Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
        inet addr:172.0.0.50  Bcast: 172.0.0.255  Mask:255.255.255.0
        [...]
```

Here `ib` typically tells us it's an InfiniBand card, but really it can be any other vendor. I have seen OmniPath using `ib` for example. Again `inet` tells us the IP of this interface is `172.0.0.50`.

If you lost me, we want the IP addresses so that we could test if ip:port is open on each node in question.

Finally, going back to our pair of `10.0.0.1:6000` and `10.0.0.2:6000` let's do an `all_reduce` test using 2 terminals, where we choose `10.0.0.1` as the master host which will coordinate other nodes. For testing we will use this helper debug program torch-distributed-gpu-test.py.

In terminal A:

```
$ ssh 10.0.0.1
$ torchrun --role $(hostname -s): --tee 3 --nnodes 2 --nproc_per_node 8 \
 --rdzv_backend c10d --rdzv_endpoint 10.0.0.1:6000 torch-distributed-gpu-test.py
```

In terminal B:

```
$ ssh 10.0.0.2
$ torchrun --role $(hostname -s): --tee 3 --nnodes 2 --nproc_per_node 8 \
 --rdzv_backend c10d --rdzv_endpoint 10.0.0.1:6000 torch-distributed-gpu-test.py
```

Note that both sides use the same `--rdzv_endpoint 10.0.0.1:6000` because we checked port 6000 is open and we use `10.0.0.1` as the coordinating host. With `c10d` rendezvous, ranks are assigned automatically, so there is no need for `--node_rank` / `--master_addr` / `--master_port`.

This approach of running things manually from each node is painful and so there are tools that automatically launch the same command on multiple nodes

pdsh

`pdsh` is one such solution - which is like `ssh` but will automatically run the same command on multiple nodes:

```
PDSH_RCMD_TYPE=ssh pdsh -w 10.0.0.1,10.0.0.2 \
'torchrun --role $(hostname -s): --tee 3 --nnodes 2 --nproc_per_node 8 \
 --rdzv_backend c10d --rdzv_endpoint 10.0.0.1:6000 torch-distributed-gpu-test.py'
```

You can see how I folded the 2 sets of commands into 1. If you have more nodes, just add more nodes as `-w` argument. (Single quotes matter here so that `$(hostname -s)` expands on each remote host rather than on the launcher.)

SLURM

If you use SLURM, it's almost certain that whoever set things up already have all the ports opened for you, so it should just work. But if it doesn't the information in this section should help debug things.

Here is how you'd use this with SLURM.

```
#!/bin/bash
#SBATCH --job-name=test-nodes        # name
#SBATCH --nodes=2                    # nodes
#SBATCH --ntasks-per-node=1          # crucial - only 1 task per dist per node!
#SBATCH --cpus-per-task=10           # number of cores per tasks
#SBATCH --gres=gpu:8                 # number of gpus
#SBATCH --time 0:05:00               # maximum execution time (HH:MM:SS)
#SBATCH --output=%x-%j.out           # output file name
#
export GPUS_PER_NODE=8
export MASTER_ADDR=$(scontrol show hostnames $SLURM_JOB_NODELIST | head -n 1)
export MASTER_PORT=6000
#
srun --jobid $SLURM_JOBID bash -c 'torchrun \
--nproc_per_node $GPUS_PER_NODE --nnodes $SLURM_NNODES --node_rank $SLURM_PROCID \
--master_addr $MASTER_ADDR --master_port $MASTER_PORT \
torch-distributed-gpu-test.py'
```

If you have more than 2 nodes you just need to change the number of nodes and the above script will automatically work

for any number of them.

MPI:

Another popular way is to use Message Passing Interface (MPI). There are a few open source implementations of it available.

To use this tool you first create a `hostfile` that contains your target nodes and the number of processes that should be run on each host. In the example of this section, with 2 nodes and 8 gpus each it'd be:

```
$ cat hostfile
10.0.0.1:8
10.0.0.2:8
```

This `host:N` hostfile syntax is the MPICH/Hydra (and Intel MPI / MVAPICH) form. Hostfiles are not standardized across MPI implementations - pick the matching line for your `mpirun`:

Implementation	Per-host count syntax
Open MPI	`10.0.0.1 slots=8`
MPICH / Hydra, Intel MPI, MVAPICH	`10.0.0.1:8`
MS-MPI	`10.0.0.1 8`

and to run, it's just:

```
$ mpirun --hostfile hostfile -np 16 -map-by ppr:8:node python my-program.py
```

Note that I used `my-program.py` here because torch-distributed-gpu-test.py was written to work with `torch.distributed.run` (also known as `torchrun`). With `mpirun` you will have to check your specific implementation to see which environment variable it uses to pass the rank of the program and replace `LOCAL_RANK` with it, the rest should be mostly the same.

Nuances:

- You might have to explicitly tell it which interface to use by adding `--mca btl_tcp_if_include 10.0.0.0/24` to match our example. If you have many network interfaces it might use one that isn't open or just the wrong interface.
- You can also do the reverse and exclude some interfaces. e.g. say you have `docker0` and `lo` interfaces - to exclude those add `--mca btl_tcp_if_exclude docker0,lo`.

`mpirun` has a gazillion of flags and I will recommend reading its manpage for more information. My intention was only to show you how you could use it. Also different `mpirun` implementations may use different CLI options.

Solving the InfiniBand connection between multiple nodes

In one situation on Azure I got 2 nodes on a shared subnet and when I tried to run the 2 node NCCL test:

```
NCCL_DEBUG=INFO python -u -m torch.distributed.run --nproc_per_node=1 --nnodes 2 --rdzv_endpoint
10.2.0.4:6000  --rdzv_backend c10d torch-distributed-gpu-test.py
```

I saw in the debug messages that InfiniBand interfaces got detected:

```
node-2:5776:5898 [0] NCCL INFO NET/IB : Using [0]ibP111p0s0:1/IB [1]rdmaP1111p0s2:1/RoCE [RO];
OOB eth0:10.2.0.4<0>
```

But the connection would then time out with the message:

```
node-2:5776:5902 [0] transport/net_ib.cc:1296 NCCL WARN NET/IB : Got completion from peer
10.2.0.5<33092> with error 12, opcode 0, len
0, vendor err 129 (Recv)
node-2:5776:5902 [0] NCCL INFO transport/net.cc:1134 -> 6
node-2:5776:5902 [0] NCCL INFO proxy.cc:679 -> 6
node-2:5776:5902 [0] NCCL INFO proxy.cc:858 -> 6 [Proxy Thread]
```

and nothing works. So here the Ethernet connectivity between 2 nodes works but not the IB interface.

There could be a variety of reason for this failing, but of the most likely one is when you're on the cloud and the 2 nodes weren't provisioned so that their IB is connected. So your Ethernet inter-node connectivity works, but it's too slow. Chances are that you need to re-provision the nodes so that they are allocated together. For example, on Azure this means you have to allocate nodes within a special availability set

Going back to our case study, once the nodes were deleted and recreated within an availability set the test worked out of the box.

The individual nodes are often not meant for inter-node communication and often the clouds have the concept of clusters, which are designed for allocating multiple nodes as a group and are already preconfigured to work together.

Prefixing logs with `node:rank`, interleaved asserts

In this section we will use `torchrun` (`torch.distributed.run`) during the demonstration and at the end of this section similar solutions for other launchers will be listed.

When you have warnings and tracebacks (or debug prints), it helps a lot to prefix each log line with its `hostname:rank` prefix, which is done by adding `--role $(hostname -s): --tee 3` to `torchrun`:

```
torchrun --role $(hostname -s): --tee 3 --nnodes 1 --nproc_per_node 2 \
torch-distributed-gpu-test.py
```

Now each log line will be prefixed with `[hostname:rank]`

Note that the colon is important.

If you're in a SLURM environment the above command line becomes:

```
srun --jobid $SLURM_JOBID bash -c 'torchrun \
--nproc_per_node $GPUS_PER_NODE --nnodes $SLURM_NNODES --node_rank $SLURM_PROCID \
--master_addr $MASTER_ADDR --master_port $MASTER_PORT \
--role $(hostname -s): --tee 3 \
torch-distributed-gpu-test.py'
```

Of course adjust your environment variables to match, this was just an example.

Important! Note, that I'm using a single quoted string of commands passed to `bash -c`. This way `hostname -s` command is delayed until it's run on each of the nodes. If you'd use double quotes above, `hostname -s` will get executed on the starting node and then all nodes will get the same hostname as the prefix, which defeats the purpose of using these flags. So if you use double quotes you need to rewrite the above like so:

```
srun --jobid $SLURM_JOBID bash -c "torchrun \
--nproc_per_node $GPUS_PER_NODE --nnodes $SLURM_NNODES --node_rank \$SLURM_PROCID \
--master_addr $MASTER_ADDR --master_port $MASTER_PORT \
--role \$(hostname -s): --tee 3 \
torch-distributed-gpu-test.py"
```

`$SLURM_PROCID` is escaped too as it needs to be specific to each node and it's unknown during the launch of the slurm job on the main node. So there are 2 `\$` escapes in this version of the command.

This prefixing functionality is also super-helpful when one gets the distributed program fail and which often results in interleaved tracebacks that are very difficult to interpret. So by `grep`ing for one `node:rank` string of choice, it's now possible to reconstruct the real error message.

For example, if you get a traceback that looks like:

```
  File "/path/to/training/dataset.py", line 785, in __init__
  File "/path/to/training/dataset.py", line 785, in __init__
    if self.dataset_proba.sum() != 1:
AttributeError: 'list' object has no attribute 'sum'
    if self.dataset_proba.sum() != 1:
  File "/path/to/training/dataset.py", line 785, in __init__
  File "/path/to/training/dataset.py", line 785, in __init__
    if self.dataset_proba.sum() != 1:
    if self.dataset_proba.sum() != 1:
AttributeError: 'list' object has no attribute 'sum'
AttributeError: 'list' object has no attribute 'sum'
AttributeError: 'list' object has no attribute 'sum'
```

and when it's dozens of frames over 8 nodes it can't be made sense of, but the above `-tee` + `--role` addition will generate:

```
[host1:0]  File "/path/to/training/dataset.py", line 785, in __init__
[host1:1]  File "/path/to/training/dataset.py", line 785, in __init__
[host1:0]    if self.dataset_proba.sum() != 1:
[host1:0]AttributeError: 'list' object has no attribute 'sum'
[host1:1]    if self.dataset_proba.sum() != 1:
[host1:2]  File "/path/to/training/dataset.py", line 785, in __init__
[host1:3]  File "/path/to/training/dataset.py", line 785, in __init__
[host1:3]    if self.dataset_proba.sum() != 1:
[host1:2]    if self.dataset_proba.sum() != 1:
[host1:1]AttributeError: 'list' object has no attribute 'sum'
[host1:2]AttributeError: 'list' object has no attribute 'sum'
[host1:3]AttributeError: 'list' object has no attribute 'sum'
```

and you can `grep` this output for just one `host:rank` prefix, which gives us:

```
$ grep -F '[host1:0]' log.txt
[host1:0]  File "/path/to/training/dataset.py", line 785, in __init__
[host1:0]    if self.dataset_proba.sum() != 1:
[host1:0]AttributeError: 'list' object has no attribute 'sum'
```

and voila, you can now tell what really happened. And as I mentioned earlier there can be easily a hundred to thousands of interleaved traceback lines there.

Also, if you have just one node, you can just pass `-tee 3` and there is no need to pass `--role`.

If `hostname -s` is too long, but you have each host with its own sequence number like:

```
[really-really-really-long-hostname-5:0]
[really-really-really-long-hostname-5:1]
[really-really-really-long-hostname-5:2]
```

you can of course make it shorter by replacing `hostname -s` with `hostname -s | tr -dc '0-9'`, which would lead to much shorter prefixes:

```
[5:0]
[5:1]
[5:2]
```

And, of course, if you're doing debug prints, then to solve this exact issue you can use `printflock`.

Here is how you accomplish the same feat with other launchers:

- `srun` in SLURM: add `--label`
- `openmpi`: add `--tag-output`
- `accelerate`: you can just pass the same `-tee` + `--role` flags as in `torchrun`

Invoke pdb on a specific rank in multi-node training

Since PyTorch 2.2 you have a handy debug feature:

```
import torch.distributed as dist
[...]

def mycode(...):

   dist.breakpoint(0)
```

This is the same as `ForkedPdb` (below) but will automatically break for you on the rank of your choice - rank0 in the example above. Just make sure to call `up;;n` right away when the breakpoint hits to get into your normal code.

Here is what it does underneath:

```
import sys
import pdb

class ForkedPdb(pdb.Pdb):
    """
    PDB Subclass for debugging multi-processed code
    Suggested in: https://stackoverflow.com/questions/4716533/
how-to-attach-debugger-to-a-python-subproccess
    """
    def interaction(self, *args, **kwargs):
        _stdin = sys.stdin
        try:
            sys.stdin = open('/dev/stdin')
            pdb.Pdb.interaction(self, *args, **kwargs)
        finally:
            sys.stdin = _stdin


def mycode():

    if dist.get_rank() == 0:
        ForkedPdb().set_trace()
    dist.barrier()
```

so you can code it yourself as well.

And you can use that `ForkedPdb` code for normal forked applications, minus the `dist` calls.

Attaching pdb to an already-running process

`dist.breakpoint()` and `ForkedPdb` above both require you to plant the breakpoint in the source before the run. Python 3.14 adds the missing case: attaching to a process that is *already running* and that you did not prepare in advance, given nothing but its PID (PEP 768):

```
python -m pdb -p PID
```

Both the target and the interpreter you attach with must be Python 3.14+, since it relies on the new safe remote-debugging hook. It drops you at a `(Pdb)` prompt controlling the target wherever it happens to be:

```
$ python -m pdb -p 2916
> /tmp/target.py(6)<module>()
-> time.sleep(0.2)
(Pdb) p counter
8
(Pdb) c
```

From there the usual commands work - `p`/`pp` to inspect, `bt` for the stack, `w` for the current frame. `c` resumes the process; closing the session detaches without killing it (in the run above the target continued and finished normally afterwards). There is no `detach` command - use `c` or quit the debugger.

This is the standard-library counterpart to `py-spy dump`: `py-spy` samples a stack read-only and never stops the target, whereas `python -m pdb -p PID` actually stops the process and lets you evaluate expressions inside it. For a wedged rank in a multi-GPU job that means you can inspect live state even though you never planted a `dist.breakpoint()`. Keep `dist.breakpoint(rank)` as the rank-coordinated path when you *can* edit the source and want a clean, all-ranks-aware stop.

A few caveats: you can normally only attach to your own processes; some systems restrict `ptrace` (the same `kernel.yama.ptrace_scope` consideration as in the py-spy section applies); the target can opt out entirely with `-X disable-remote-debug` or `PYTHON_DISABLE_REMOTE_DEBUG=1`; and a rank blocked inside a C call will not break until that call returns.

Primary reference: Python 3.14 pdb.

Performance and profiling

Measuring durations

One of the key things we care about is in ML software is its speed. This is because when the program runs faster:

- we need to spend less money on the hardware (even if you own the hardware, you still need to pay electricity to run it).
- the program finishes faster - have you heard of the AI race? You can have the best idea for a new model but if the competitor's framework trains the model faster that yours your idea could become irrelevant by the time your software completes the training.

Well, may be add bragging rights to the list above ;)

The first need of making software faster is measuring various runtime durations, then optimizing, measuring again and hopefully getting some of those runtime durations shorter.

time

So obviously the first tool to reach out is time.

```
$ python -c 'import time; start=time.time(); time.sleep(1); print(f"duration: {time.time()-start}
secs")'
duration: 1.0000815391540527 secs
```

The little one-liner took a snapshot of start time, then slept for 1 seconds and then took another time snapshot and reported the difference. That's all.

As you may have noticed in the earlier sections of this chapter that there are different types of time.

```
$ time python -c 'import torch'
real    0m1.399s
user    0m8.222s
sys     0m0.209s
```

Here:

- `real` signifies the wall clock time - that is if you were to use a stop-watch - this the amount of elapsed time since you launched the program and it has finished its run
- `user` is the amount of time the program spent performing calls in the user-space - that is your program
- `sys` is the amount of time performing system calls (operating system / kernel level) - things like IO, networking, memory

Here `user+sys` (`8.222+0.209=8.431`) is six times bigger than `real` (`1.399`). That's not a contradiction: `real` is what a stop-watch would show, while `user` and `sys` count cpu-time summed over every thread that ran. This compute node has 192 cpu-cores and importing `torch` spins up a thread pool across them, so cpu-seconds pile up much faster than wall clock seconds. Restrict it to a single thread and the cpu-time collapses while the wall clock time stays where it was:

```
$ time OMP_NUM_THREADS=1 python -c 'import torch'
real    0m1.377s
user    0m1.099s
sys     0m0.258s
```

Now `user+sys` (`1.099+0.258=1.357`) does add up to `real` (`1.377`) - and note that the 7 extra cpu-seconds of the first run bought no wall clock time at all. `real` time can also be bigger than `user+sys`, and that happens when the program waits instead of computing. If, for example, you're doing some very demanding compilation like building PyTorch using `make -j` to use all the cpu-cores, and at the same time you decide to run another program on the same system - the latter is likely to report a bigger `real` time and a smaller `user` plus `sys` time, because your program will be fighting with dozens of copies of `gcc` to get its share of a cpu time, spending a lot of time waiting. Slow or blocking IO (e.g. shared nfs filesystem) is another example of the same discrepancy. Usually if your software isn't competing with another software, then `real` is all you should care about. Analyzing `sys` + `user` is important for those who optimize the low level systems.

hyperfine

hyperfine runs a given command multiple times and reports how long it took to run: the mean, the standard deviation, and the fastest and slowest run. It can also compare two or more commands and report which one is faster and by how much.

If you don't already have it installed do:

```
sudo apt-get install -y hyperfine
```

Here it is on the same import that was measured with `time`:

```
$ hyperfine 'python -c "import torch"'
Benchmark 1: python -c "import torch"
  Time (mean ± σ):      1.412 s ±  0.024 s    [User: 8.160 s, System: 0.290 s]
  Range (min … max):    1.378 s …  1.454 s    10 runs
```

It ran the command 10 times, which is the default - use `-r N` to set the number of runs. The `± 0.024 s` is the part `time` can't tell you: repeated runs of the same command differ from each other by about 24ms, so a change that saves 10ms hasn't been shown to save anything.

To compare two commands, pass both of them:

```
$ hyperfine -w 2 'python -c "import torch"' 'python -c "import transformers"'
Benchmark 1: python -c "import torch"
  Time (mean ± σ):      1.415 s ±  0.026 s    [User: 8.226 s, System: 0.224 s]
  Range (min … max):    1.379 s …  1.455 s    10 runs

Benchmark 2: python -c "import transformers"
  Time (mean ± σ):      1.211 s ±  0.017 s    [User: 8.024 s, System: 0.245 s]
  Range (min … max):    1.186 s …  1.233 s    10 runs

Summary
  python -c "import transformers" ran
    1.17 ± 0.03 times faster than python -c "import torch"
```

`-w 2` performs a warmup by running each command twice before it starts measuring. Importing `transformers` is faster than importing `torch`, even though `transformers` needs `torch`, because it doesn't import it:

```
$ python -c 'import transformers, sys; print("torch" in sys.modules)'
False
```

Every measurement above most likely came from the cache - the files being imported were already in the kernel's page cache, put there by earlier runs. When they aren't cached they get read off the disk instead, which can happen for any number of reasons - a new node, a fresh install, the first run of the day, someone flushing the cache or the kernel reclaiming that memory for something else. Use `--prepare` or `-p` to run a command before each timed run, and drop the caches there:

```
$ hyperfine -p 'sync; echo 3 | sudo tee /proc/sys/vm/drop_caches' 'python -c "import torch"'
Benchmark 1: python -c "import torch"
  Time (mean ± σ):      2.103 s ±  0.054 s    [User: 8.319 s, System: 0.348 s]
  Range (min … max):    2.018 s …  2.191 s    10 runs
```

The same import takes 1.5x longer with no cache to read from.

It is likely to take much longer than that if the files are on a networked filesystem. Here is the `torch` package copied onto a local NVMe drive and onto a shared Lustre mount, then imported from each:

package on	from cache	from the filesystem
local NVMe	1.491s ± 0.178	2.134s ± 0.035
Lustre	1.681s ± 0.494	10.589s ± 0.104

With nothing cached, Lustre takes 5.0x as long as the local drive, and 7.1x as long as the local drive reading from cache. This is why installing packages on a shared filesystem and then importing them is so much slower than doing the same thing on a local disk. Writing the package is punished even harder than reading it - copying these same files took 2.026s onto the local drive and 2m35.880s onto Lustre. The cached Lustre figure is unreliable as well - those runs ranged from 1.441s to 2.912s, because the Lustre client doesn't keep a package that size cached for long.

`User` time stays at about 8.3s in all of these runs, so the extra wall clock time is time spent waiting for IO.

What an import actually reads, and how to build a file system benchmark out of timings like these, is covered in Usability perception IO benchmarks.

So when you compare two timings, make sure both were taken the same way - both from cache or both not, and both from the same filesystem. Otherwise what you're measuring is the cache and the filesystem, not the change you made.

Event-based durations

When measuring code execution on GPUs (and another non-CPU devices), which always involves launching CUDA and similar kernels, using wall clock time won't give you the correct measurement, which means you won't be able to reliably optimize your kernels or code using those kernels. The correct way to do it precisely is to use events. In the following example we use CUDA events:

```
import torch
start_event = torch.cuda.Event(enable_timing=True)
end_event   = torch.cuda.Event(enable_timing=True)
x = torch.ones(2**10,2**10, device="cuda")
start_event.record()
x@x.T
end_event.record()
torch.cuda.synchronize()
duration = start_event.elapsed_time(end_event) / 1000
print(f"It took {duration} seconds to run this")
```

It will report:

```
It took 0.07701897430419923 seconds to run this
```

The code creates 2 CUDA event objects, then it creates a large tensor, starts recording on the first event, runs a huge matmul of that tensor with the transposed version of itself, records the end of the event, waits till any asyncronious events complete and then measures the duration of time between the two events. `start_event.elapsed_time(end_event)` can be called any time later. The important exact sequence/order is just these lines:

```
start_event.record()
<the code execution of which you want to time goes here>
end_event.record()
torch.cuda.synchronize()
```

If you're working with other devices, most of them have similar or identical API:

- `torch.cuda.Event` - NVIDIA CUDA + AMD ROCm
- `ht.hpu.Event(enable_timing)` - Intel Gaudi
- `torch.xpu.Event(enable_timing)` - Intel dGPUs
- Apple MPS has a different API, but we can emulate the same API as can seen in mamf-finder.py.

Here are good practical examples of measuring time durations with the help of device events:

- mamf-finder
- all_reduce_bench

Here are some excellent articles going into deeper explanations and examples:

- How to Accurately Time CUDA Kernels in Pytorch

Profilers

In Python Profilers we have seen how to use general Python profilers to detect where the program is slow. For PyTorch there are specially designed profilers that give you a better view on the PyTorch APIs and enables you to work not only with CPUs but with GPUs, TPUs, and other hardware. These are designed to give a very small overheads and provide access to fine grained APIs not visible from Python profilers.

Since PyTorch operations are usually performed asynchronously the regular profilers like cProfile will not be able to provide correct measurements since it doesn't know how long the actual kernel operation has taken and it'll instead report something that it can see that happens on CPU.

torch.profiler

`torch.profiler` allows one to profile PyTorch API, correctly measuring the actual execution time which often is done asynchronously - that is a Python dispatchers pushes the kernel execution command into a queue and then when the scheduler sees an opportunity the kernel gets executed. This functionality cannot be performed with Python profilers like cProfile.

Let's look at a simple example:

```
# torch-profile-linear-example.py
import torch
from torch.profiler import profile, ProfilerActivity

linear = torch.nn.Linear(1024, 1024).to("cuda")
x = torch.randn(256, 512, 1024, device="cuda")

# warmup
out = linear(x)
torch.cuda.synchronize()

with profile(activities=[ProfilerActivity.CUDA], with_stack=True) as prof:
    out = linear(x)
    torch.cuda.synchronize()
print(prof.key_averages().table(sort_by="cuda_time_total", row_limit=10,
max_name_column_width=27))
```

It creates a Linear layer, creates a random input tensor and feeds it to the Linear layer. This is done twice, the first time is the warmup which we ignore and the second time we run the linear layer with `torch.profiler` and then print the profiler report:

```
$ python torch-profile-linear-example.py
---------------------------  ----------  ---------  -----------  ---------  ------------
---------  -----------  ----------  -------------  ----------
                      Name  Self CPU %   Self CPU  CPU total %  CPU total  CPU time avg  Self
CUDA  Self CUDA %  CUDA total  CUDA time avg  # of Calls
---------------------------  ----------  ---------  -----------  ---------  ------------
---------  -----------  ----------  -------------  ----------
```

```
sm90_xmma_gemm_f32f32_tf...       0.00%    0.000us        0.00%    0.000us       0.000us
687.139us       99.80%   687.139us      687.139us           1
            Memset (Device)       0.00%    0.000us        0.00%    0.000us       0.000us
1.344us        0.20%     1.344us        1.344us           1
    Activity Buffer Request      69.98%    2.193ms       69.98%    2.193ms       2.193ms
0.000us        0.00%     0.000us        0.000us           1
      cudaStreamIsCapturing       0.26%    8.203us        0.26%    8.203us       8.203us
0.000us        0.00%     0.000us        0.000us           1
                cudaMalloc      10.75%  337.029us       10.75%  337.029us     337.029us
0.000us        0.00%     0.000us        0.000us           1
           cudaMemsetAsync       0.61%   19.135us        0.61%   19.135us      19.135us
0.000us        0.00%     0.000us        0.000us           1
      cudaFuncGetAttributes       0.28%    8.769us        0.28%    8.769us       4.385us
0.000us        0.00%     0.000us        0.000us           2
        cudaLaunchKernelExC       1.23%   38.672us        1.23%   38.672us      38.672us
0.000us        0.00%     0.000us        0.000us           1
      cudaDeviceSynchronize      16.88%  529.079us       16.88%  529.079us     264.540us
0.000us        0.00%     0.000us        0.000us           2
---------------------------  ----------  ---------  -----------  ---------  ------------
---------  -----------  ----------  -------------  ----------
Self CPU time total: 3.134ms
Self CUDA time total: 688.483us
```

From the report it's easy to see that the code spends most of its time performing a GEMM operation, since that's what a linear layer does. You can see that it took 687.139us, which accounted for 99.80% of CUDA operations of this run. The rest of the calls are various CUDA functions that were used while launching the GEMM operation `sm90_xmma_gemm_f32f32_tf...`.

Since kernel names tend to include dtype and shapes they can be pretty long, so if you'd like to see the full name, you can make `max_name_column_width` longer, for example:

```
- print(prof.key_averages().table(sort_by="cuda_time_total", row_limit=10,
max_name_column_width=27))
+ print(prof.key_averages().table(sort_by="cuda_time_total", row_limit=10,
max_name_column_width=450))
```

The other interesting parts of the report are the end-to-end timings:

```
Self CPU time total: 3.134ms
Self CUDA time total: 688.483us
```

thus you can see that in total about 3.1 msec were spent on CPU and 0.7 msec on CUDA.

Of course, once you profile a whole model, rather than a single Linear layer you will see a lot more kernels in the profiler output and that's where things become interesting.

For more options please refer to the `torch.profiler` doc.

Additionally, here is an excellent introduction to `torch.profiler` from the HuggingFace team.

When torch.profiler isn't enough

In the introduction it was stated that cProfile is the wrong profiler for PyTorch code, however there are situations where you want to use cProfile with PyTorch code.

Recently I have been diagnosing a strange ~1 sec overhead in forward and backward calls, the `torch.profiler` forward measurement would take about 100 msec but the total wallclock timer would be around 1 sec. I was getting no help from torch.profiler and decided to run cProfile instead. I immediately saw the issue - it was a triton kernel recompilation that was taking about 1 sec. As I was working with a flattened 2D padded input into 1D unpadded input, the final unpadded tensors was different on many steps and was triggering a kernel recompilation which was written to work with specific input length.

Let's reproduce this use case and work with different debug tools to understand the situation.

I assume you already have the correct version of torch installed, if not head here.

We just need `liger-kernel` installed then:

```
pip install liger-kernel
```

Let's first write a flexible profiler context manager that allows us to quickly switch between cProfile, `torch.profiler` and no profiler at all.

```
# profilers.py
from contextlib import nullcontext
from pstats import Stats
from torch.profiler import profile, record_function, ProfilerActivity
import cProfile
import pstats
import torch

# customize the precision of cProfile to give 6 decimals
pstats.f8 = lambda x: f"{x:3.6f}"


class ProfilerContext:
    """
    A proxy Profiler context manager class that can quickly choose between cProfile,
torch.profiler and no-profiler w/o changing the end user code (other than changing the profiler
type flag)

    Example:

    prof_fwd = ProfilerContext(type="c", name="some context")
    with prof_fwd():
        x = 1
    prof_fwd.report()
    """

    def __init__(self, type="none", name=None):
        """
        Args:
```

```
        - type: "torch": torch.profiler, "c": cProfile, "none": none
        - name: some context string for the reports
        """
        self.torch = False
        self.c = False
        if type == "torch":
            self.torch = True
        elif type == "c":
            self.c = True
        elif type == "none":
            pass
        else:
            raise ValueError(f"the `type` can be one of torch|c|none but got {type}")

        self.name = name if name is not None else "unknown"

        if self.torch:
            self.ctx = profile(activities=[ProfilerActivity.CUDA], record_shapes=False,
with_stack=True)
        elif self.c:
            self.ctx = cProfile.Profile()
        else:
            self.ctx = nullcontext()

    def __call__(self):
        if self.torch and torch.cuda.is_available():
            torch.cuda.synchronize()

        return self.ctx

    def report(self):
        if self.torch:
            print(f"*** torch.profiler {self.name} ***")
            print(self.ctx.key_averages().table(sort_by="cuda_time_total", row_limit=20))
        elif self.c:
            print(f"*** cProfile {self.name} ***")
            stats = Stats(self.ctx)
            stats.sort_stats("tottime").print_stats(20)
            # cumulative report is useful to understand where some of the large internal time
overheads
            # come from - because it shows you the stack of calls leading to the slow call. So
            # `tottime` shows candidates to study and `cumulative` for finding context for those
calls
            stats.sort_stats('cumulative').print_stats(50)
        else:
            pass # report nothing
```

Now let's use it to profile 2 forward calls using Liger Kernel:

```
# liger-kernel-varlen-recompile.py
```

```
import torch
import time
from liger_kernel.transformers import AutoLigerKernelForCausalLM
from profilers import ProfilerContext

#PROFILER_TYPE = "c"
PROFILER_TYPE = "torch"
#PROFILER_TYPE = "none"

model = AutoLigerKernelForCausalLM.from_pretrained("Qwen/Qwen3-0.6B").to("cuda")

batch_size = 5
seq_len = 512
vocab_size = model.config.vocab_size

input_ids = torch.randint(0, vocab_size, (batch_size, seq_len), device=model.device)
attention_mask = torch.ones_like(input_ids)

for i in range(2):
    prof = ProfilerContext(type=PROFILER_TYPE, name=f"FWD {i}")
    start = time.perf_counter()
    with prof(): outputs = model(input_ids=input_ids, attention_mask=attention_mask)
    prof.report()
    print(f"wallclock duration: {(time.perf_counter()-start)*1000:0.3f} msecs")
```

When we run it we get this discrepancy I mentioned earlier:

```
$ python liger-kernel-varlen-recompile.py
*** torch.profiler FWD 0 ***
[...]
Self CPU time total: 119.385ms
Self CUDA time total: 25.041ms

wallclock duration: 1003.727 msecs
```

You can see that the difference between CPU/CUDA time reports and the measured wallclock duration is huge, but on the second step the difference is much smaller:

```
*** torch.profiler FWD 1 ***
[...]
Self CPU time total: 59.184ms
Self CUDA time total: 25.011ms

wallclock duration: 228.978 msecs
```

I trimmed out most of the profiling report to show just the relevant for this discussion parts.

Now why does the first forward call takes much longer than PyTorch's `forward` call? Is it because something happens that

is not PyTorch related? So let's use the same script but switch from `torch.profiler` to `cProfile`, but just editing the script to:

```
PROFILER_TYPE = "c"
#PROFILER_TYPE = "torch"
```

Now when we rerun the script, we quickly see the issue on the first forward call:

```
*** cProfile FWD 0 ***
         113616 function calls (109641 primitive calls) in 3.290 seconds

   Ordered by: internal time
   List reduced from 947 to 20 due to restriction <20>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       28 1.991778 0.071135 1.991778 0.071135 {built-in method
torch._C._nn.scaled_dot_product_attention}
       57 0.427335 0.007497 0.427335 0.007497 {built-in method torch.cat}
      398 0.268139 0.000674 0.268139 0.000674 {method 'update' of '_hashlib.HASH' objects}
      197 0.134879 0.000685 0.134879 0.000685 {built-in method torch._C._nn.linear}
       28 0.107747 0.003848 0.107747 0.003848 {built-in method torch.empty_like}
      475 0.091337 0.000192 0.091337 0.000192 {method 'read' of '_io.BufferedReader' objects}
        1 0.050481 0.050481 0.055220 0.055220 python3.12/site-packages/transformers/models/qwen3/
modeling_qwen3.py:319(forward)
       80 0.035789 0.000447 0.036184 0.000452 {built-in method _io.open}
      115 0.025036 0.000218 0.025036 0.000218 {method '__exit__' of '_io._IOBase' objects}
    144/7 0.008807 0.000061 0.037330 0.005333 {built-in method builtins.exec}
        6 0.005818 0.000970 0.005818 0.000970 {built-in method _imp.create_dynamic}
        1 0.005175 0.005175 0.005175 0.005175 {method 'all' of 'torch._C.TensorBase' objects}
      376 0.004812 0.000013 0.004812 0.000013 {built-in method posix.stat}
```

One can instantly see that there are quite a few calls that are clearly from the libc library and which interface with the system, e.g., the `exec`, `posix.stat` and `io.open` calls. This is definitely something we don't want to happen during the super fast `forward`/`backward` calls using PyTorch.

The cProfile wrapper I wrote dumps the profiler data twice, first sorted by internal time which helps us quickly see which methods are slow in general - but that doesn't help us understand where these calls came from, so the second dump sorts by accumulation time and helps us discover the caller stack trace. So if we look at the second dump:

```
         113616 function calls (109641 primitive calls) in 3.290 seconds

   Ordered by: cumulative time
   List reduced from 947 to 50 due to restriction <50>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
[...]]
      141 0.000894 0.000006 0.620426 0.004400 /home/yak/miniconda3/envs/dev/lib/python3.12/
site-packages/liger_kernel/ops/utils.py:33(wrapper)
```

```
      169 0.000906 0.000005 0.524757 0.003105 /home/yak/miniconda3/envs/dev/lib/python3.12/
site-packages/triton/runtime/jit.py:370(<lambda>)
      169 0.004586 0.000027 0.523851 0.003100 /home/yak/miniconda3/envs/dev/lib/python3.12/
site-packages/triton/runtime/jit.py:695(run)
      113 0.000454 0.000004 0.498234 0.004409 /home/yak/miniconda3/envs/dev/lib/python3.12/
site-packages/liger_kernel/transformers/rms_norm.py:37(forward)
      113 0.000670 0.000006 0.490840 0.004344 /home/yak/miniconda3/envs/dev/lib/python3.12/
site-packages/liger_kernel/ops/rms_norm.py:605(forward)
      113 0.002546 0.000023 0.490049 0.004337 /home/yak/miniconda3/envs/dev/lib/python3.12/
site-packages/liger_kernel/ops/rms_norm.py:409(rms_norm_forward)
        5 0.000087 0.000017 0.435603 0.087121 /home/yak/miniconda3/envs/dev/lib/python3.12/
site-packages/triton/runtime/jit.py:826(_do_compile)
        5 0.000169 0.000034 0.435442 0.087088 /home/yak/miniconda3/envs/dev/lib/python3.12/
site-packages/triton/compiler/compiler.py:226(compile)
       57 0.427335 0.007497 0.427335 0.007497 {built-in method torch.cat}
       28 0.000101 0.000004 0.426423 0.015229 /home/yak/miniconda3/envs/dev/lib/python3.12/
site-packages/transformers/cache_utils.py:742(update)
       28 0.000226 0.000008 0.426321 0.015226 /home/yak/miniconda3/envs/dev/lib/python3.12/
site-packages/transformers/cache_utils.py:98(update)
        5 0.000105 0.000021 0.368456 0.073691 /home/yak/miniconda3/envs/dev/lib/python3.12/
site-packages/triton/runtime/cache.py:307(get_cache_key)
        1 0.000693 0.000693 0.349386 0.349386 /home/yak/miniconda3/envs/dev/lib/python3.12/
site-packages/triton/runtime/cache.py:271(triton_key)
      398 0.268139 0.000674 0.268139 0.000674 {method 'update' of '_hashlib.HASH' objects}
```

We can quickly understand that the weird libc calls came from Liger Kernel using Triton, which in turn compiles the kernel and caches it. Hopefully it's obvious from the stack trace above.

Now, normally it's perfectly fine that the first call is likely to run some optimizations (e.g. `torch.compile`) which could take longer than the subsequent calls, but in case of the older versions of liger-kernel there was a bug that recompiled and cached the `RMSNorm` kernel for every new sequence length, which massively impacted the end-to-end performance (moreover it'd do it twice for `forward` and `backward` since those are 2 different kernels).

So if you install `pip install liger-kernel==0.6.1` you will see this problem if your sequence length changes from step to step. I found that installing `liger-kernel>=0.8.0` fixes the problem.

In general when you benchmark code you need a warmup phase where the code is exercised first and you start benchmarking things after step 2 or even later at times, but in this case I wanted to demonstrate how cProfile can still be useful when you profile seemingly pure PyTorch code and you observe that it's underperforming, and it was enough to do it in the very first step. But it'd work just as fine in step 2 and onwards.

footnote: since the culprit here was `torch.compile`, you could also have spotted the recompilation directly with `TORCH_LOGS=recompiles` (and `TORCH_LOGS=graph_breaks` for eager fallbacks); see PyTorch compiler troubleshooting.

Debug Tools

git-related tools

Useful aliases

Show a diff of all files modified in the current branch against the remote default branch:

```
alias brdiff="def_branch=\$(git symbolic-ref refs/remotes/origin/HEAD | sed 's@^refs/remotes/
origin/@@'); git diff origin/\$def_branch..."
```

Same, but ignore white-space differences, adding `--ignore-space-at-eol` or `-w`:

```
alias brdiff-nows="def_branch=\$(git symbolic-ref refs/remotes/origin/HEAD | sed 's@^refs/
remotes/origin/@@'); git diff -w origin/\$def_branch..."
```

List all the files that were added or modified in the current branch compared to the remote default branch:

```
alias brfiles="def_branch=\$(git symbolic-ref refs/remotes/origin/HEAD | sed 's@^refs/remotes/
origin/@@'); git diff --name-only origin/\$def_branch..."
```

Once we have the list, we can now automatically open an editor to load just added and modified files:

```
alias bremacs="def_branch=\$(git symbolic-ref refs/remotes/origin/HEAD | sed 's@^refs/remotes/
origin/@@'); emacs \$(git diff --name-only origin/\$def_branch...) &"
```

git-bisect

(note to self: this is a sync from `the-art-of-debugging/methodology.md` which is the true source)

The discussed next approach should work for any revision control system that supports bisecting. We will use `git bisect` in this discussion.

`git bisect` helps to quickly find the commit that caused a certain problem.

Use case: Say, you were using `transformers==4.33.0` and then you needed a more recent feature so you upgraded to the bleed-edge `transformers@main` and your code broke. There could have been hundreds of commits between the two versions and it'd be very difficult to find the right commit that lead to the breakage by going through all the commits. Here is how you can quickly find out which commit was the cause.

footnote: HuggingFace Transformers is actually pretty good at not breaking often, but given its complexity and enormous size it happens nevertheless and the problems are fixed very quickly once reported. Since it's a very popular Machine Learning library it makes for a good debugging use case.

Solution: Bisecting all the commits between the known good and bad commits to find the one commit that's to blame.

We are going to use 2 shell terminals: A and B. Terminal A will be used for `git bisect` and terminal B for testing your software. There is no technical reason why you couldn't get away with a single terminal but it's easier with 2.

1. In terminal A fetch the git repo and install it in devel mode (`pip install -e .`) into your Python environment.

```
git clone https://github.com/huggingface/transformers
cd transformers
pip install -e .
```

Now the code of this clone will be used automatically when you run your application, instead of the version you previously installed from PyPi or Conda or elsewhere.

Also for simplicity we assume that all the dependencies have already been installed.

2. next we launch the bisecting - In terminal A, run:

```
git bisect start
```

3. Discover the last known good and the first known bad commits

`git bisect` needs just 2 data points to do its work. It needs to know one earlier commit that is known to work (`good`) and one later commit that is known to break (`bad`). So if you look at the sequence of commits on a given branch it'd have 2 known points and many commits around these that are of an unknown quality:

```
...... orig_good ..... .... .... .... ..... orig_bad ....
------------>---------------->----------------> time
```

So for example if you know that `transformers==4.33.0` was good and `transformers@main` (`HEAD`) is bad, find which commit is corresponding to the tag `4.33.0` by visiting the releases page and searching for `4.33.0`. We find that it was commit with SHA `5a4f340d`.

footnote: typically the first 8 hex characters are enough to have a unique identifier for a given repo, but you can use the full 40 character string.

So now we specify which is the first known good commit:

```
git bisect good 5a4f340d
```

and as we said we will use `HEAD` (latest commit) as the bad one, in which case we can use `HEAD` instead finding out the corresponding SHA string:

```
git bisect bad HEAD
```

If however you know it broke in `4.34.0` you can find its latest commit as explained above and use that instead of `HEAD`.

We are now all set at finding out the commit that broke things for you.

And after you told `git bisect` the good and the bad commits it has already switched to a commit somewhere in the middle:

```
...... orig_good ..... .... current .... .... ..... orig_bad ........
------------>--------------->---------------->----------------> time
```

You can run `git log` to see which commit it has switched to.

And to remind, we installed this repo as `pip install -e .` so the Python environment is instantly updated to the current commit's code version.

4. Good or bad

The next stage is telling `git bisect` if the current commit is `good` or `bad`:

To do so in terminal B run your program once.

Then in terminal A run:

```
git bisect bad
```

If it fails, or:

```
git bisect good
```

if it succeeds.

If, for example, if the result was bad, `git bisect` will internally flag the last commit as new bad and will half the commits again, switching to a new current commit:

```
...... orig_good ..... current .... new_bad .... ..... orig_bad ....
------------>--------------->---------------->----------------> time
```

And, vice versa, if the result was good, then you will have:

```
...... orig_good ..... .... new_good .... current ..... orig_bad ....
------------>--------------->---------------->----------------> time
```

5. Repeat until no more commits left

Keep repeating step 4 step until the problematic commit is found.

Once you finished bisecting, `git bisect` will tell you which commit was responsible for breaking things.

```
...... orig_good ..... .... last_good first_bad .... .. orig_bad ....
------------>--------------->---------------->----------------> time
```

If you followed the little commit diagrams, it'd correspond for the`first_bad` commit.

You can then go to `https://github.com/huggingface/transformers/commit/` and append the commit SHA to that url which will take you to the commit, (e.g. `https://github.com/huggingface/transformers/commit/`

`57f44dc4288a3521bd700405ad41e90a4687abc0` and which will then link to the PR from which it originated. And then you can ask for help by following up in that PR.

If your program doesn't take too long to run even if there are thousands of commits to search, you are facing `n` bisecting steps from `2**n` so 1024 commits can be searched in 10 steps.

If your program is very slow, try to reduce it to something small - ideally a small reproduction program that shows the problem really fast. Often, commenting out huge chunks of code that you deem irrelevant to the problem at hand, can be all it takes.

If you want to see the progress, you can ask it to show the current range of remaining commits to check with:

```
git bisect visualize --oneline
```

6. Clean up

So now restore the git repo clone to the same state you started from (most likely `HEAD`) with:

```
git bisect reset
```

and possible reinstall the good version of the library while you report the issue to the maintainers.

Sometimes, the issue emerges from intentional backward compatibility breaking API changes, and you might just need to read the project's documentation to see what has changed. For example, if you switched from `transformers==2.0.0` to `transformers==3.0.0` it's almost guaranteed that your code will break, as major numbers difference are typically used to introduce major API changes.

7. Possible problems and their solutions:

a. skipping

If for some reason the current commit cannot be tested - it can be skipped with:

```
git bisect skip
```

and `git bisect` will continue bisecting the remaining commits.

This is often helpful if some API has changed in the middle of the commit range and your program starts to fail for a totally different reason.

You might also try to make a variation of the program that adapts to the new API, and use it instead, but it's not always easy to do.

b. reversing the order

Normally git expects `bad` to be after `good`.

```
...... orig_good ..... .... .... .... ..... orig_bad ....
------------>--------------->---------------->----------------> time
```

Now, if `bad` happens before `good` revision order-wise and you want to find the first revision that fixed a previously existing problem - you can reverse the definitions of `good` and `bad` - it'd be confusing to work with overloaded logic states, so it's

recommended to use a new set of states instead - for example, `fixed` and `broken` - here is how you do that.

```
git bisect start --term-new=fixed --term-old=broken
git bisect fixed
git bisect broken 6c94774
```

and then continue marking commits with:

```
git bisect fixed
git bisect broken
```

instead of:

```
git bisect good
git bisect bad
```

c. complications

There are sometimes other complications, like when different revisions' dependencies aren't the same and for example one revision may require `numpy=1.25` and the other `numpy=1.26`. If the dependency package versions are backward compatible installing the newer version should do the trick. But that's not always the case. So sometimes one has to reinstall the right dependencies before re-testing the program.

Sometimes, it helps when there is a range of commits that are actually broken in a different way, you can either find a range of `good...bad` commits that isn't including the other bad range, or you can try to `git bisect skip` the other bad commits as explained earlier.

A Backup of Scripts

This is a backup of scripts discussed in Faster debug and development with tiny models, tokenizers and datasets.

- c4-en-10k.py
- cm4-synthetic-testing.py
- fsmt-make-super-tiny-model.py
- general-pmd-ds-unpack.py
- general-pmd-synthetic-testing.py
- m4-ds-unpack.py
- mt5-make-tiny-model.py
- openwebtext-10k.py
- oscar-en-10k.py

Writing and Running Tests

Note: a part of this document refers to functionality provided by the included testing_utils.py, the bulk of which I have developed while I worked at HuggingFace.

This document covers both `pytest` and `unittest` functionalities and shows how both can be used together.

Running tests

Run all tests

```
pytest
```

I use the following alias:

```
alias pyt="pytest --disable-warnings --instafail -rA"
```

which tells pytest to:

- disable warning
- `--instafail` shows failures as they happen, and not at the end
- `-rA` generates a short test summary info

it requires you to install:

```
pip install pytest-instafail
```

Getting the list of all tests

Show all tests in the test suite:

```
pytest --collect-only -q
```

Show all tests in a given test file:

```
pytest tests/test_optimization.py --collect-only -q
```

I use the following alias:

```
alias pytc="pytest --disable-warnings --collect-only -q"
```

Run a specific test module

To run an individual test module:

```
pytest tests/utils/test_logging.py
```

Run specific tests

If `unittest` is used, to run specific subtests you need to know the name of the `unittest` class containing those tests. For example, it could be:

```
pytest tests/test_optimization.py::OptimizationTest::test_adam_w
```

Here:

- `tests/test_optimization.py` - the file with tests
- `OptimizationTest` - the name of the test class
- `test_adam_w` - the name of the specific test function

If the file contains multiple classes, you can choose to run only tests of a given class. For example:

```
pytest tests/test_optimization.py::OptimizationTest
```

will run all the tests inside that class.

As mentioned earlier you can see what tests are contained inside the `OptimizationTest` class by running:

```
pytest tests/test_optimization.py::OptimizationTest --collect-only -q
```

You can run tests by keyword expressions.

To run only tests whose name contains `adam`:

```
pytest -k adam tests/test_optimization.py
```

Logical `and` and `or` can be used to indicate whether all keywords should match or either. `not` can be used to negate.

To run all tests except those whose name contains `adam`:

```
pytest -k "not adam" tests/test_optimization.py
```

And you can combine the two patterns in one:

```
pytest -k "ada and not adam" tests/test_optimization.py
```

For example to run both `test_adafactor` and `test_adam_w` you can use:

```
pytest -k "test_adafactor or test_adam_w" tests/test_optimization.py
```

Note that we use `or` here, since we want either of the keywords to match to include both.

If you want to include only tests that include both patterns, `and` is to be used:

```
pytest -k "test and ada" tests/test_optimization.py
```

Run only modified tests

You can run the tests related to the unstaged files or the current branch (according to Git) by using pytest-picked. This is a great way of quickly testing your changes didn't break anything, since it won't run the tests related to files you didn't touch.

```
pip install pytest-picked
```

```
pytest --picked
```

All tests will be run from files and folders which are modified, but not yet committed.

Automatically rerun failed tests on source modification

pytest-xdist provides a very useful feature of detecting all failed tests, and then waiting for you to modify files and continuously re-rerun those failing tests until they pass while you fix them. So that you don't need to re start pytest after you made the fix. This is repeated until all tests pass after which again a full run is performed.

```
pip install pytest-xdist
```

To enter the mode: `pytest -f` or `pytest --looponfail`

File changes are detected by looking at `looponfailroots` root directories and all of their contents (recursively). If the default for this value does not work for you, you can change it in your project by setting a configuration option in `setup.cfg`:

```
[tool:pytest]
looponfailroots = transformers tests
```

or `pytest.ini`/`tox.ini` files:

```
[pytest]
looponfailroots = transformers tests
```

This would lead to only looking for file changes in the respective directories, specified relatively to the ini-file's directory.

pytest-watcher is an alternative implementation of this functionality. The older `pytest-watch` is widely linked but has had no commit since 2022, so prefer the former or `--looponfail` above.

Skip a test module

If you want to run all test modules, except a few you can exclude them by giving an explicit list of tests to run. For example, to run all except `test_modeling_*.py` tests:

```
pytest $(ls -1 tests/*py | grep -v test_modeling)
```

Clearing state

CI builds and when isolation is important (against speed), cache should be cleared:

```
pytest --cache-clear tests
```

Running tests in parallel

If your project defines a `make test` target, it typically runs tests in parallel via the `pytest-xdist` plugin (`-n X` argument, e.g. `-n 2` to run 2 parallel jobs).

`pytest-xdist`'s `--dist=` option allows one to control how the tests are grouped. `--dist=loadfile` puts the tests located in one file onto the same process.

Since the order of executed tests is different and unpredictable, if running the test suite with `pytest-xdist` produces failures (meaning we have some undetected coupled tests), use pytest-replay to replay the tests in the same order, which should help with then somehow reducing that failing sequence to a minimum.

Test order and repetition

It's good to repeat the tests several times, in sequence, randomly, or in sets, to detect any potential inter-dependency and state-related bugs (tear down). And the straightforward multiple repetition is just good to detect some problems that get uncovered by randomness of DL.

Repeat tests

- pytest-flakefinder:

```
pip install pytest-flakefinder
```

And then run every test multiple times (50 by default):

```
pytest --flake-finder --flake-runs=5 tests/test_failing_test.py
```

footnote: This plugin doesn't work with `-n` flag from `pytest-xdist`. It has also had no commit since 2022, so treat it as functional-but-unmaintained rather than actively developed.

footnote: There is another plugin `pytest-repeat`, which is actively maintained but cannot repeat `unittest.TestCase`

tests - they "simply always run once, regardless of `--count`" and emit a warning. So the two swap trade-offs: `pytest-flakefinder` handles `unittest` classes but is dormant, `pytest-repeat` is current but skips them silently apart from that warning. If your suite is built on `unittest.TestCase`, as many `transformers`-derived suites are, that limitation decides it.

Run tests in a random order

```
pip install pytest-random-order
```

Important: the presence of `pytest-random-order` will automatically randomize tests, no configuration change or command line options is required.

As explained earlier this allows detection of coupled tests - where one test's state affects the state of another. When `pytest-random-order` is installed it will print the random seed it used for that session, e.g:

```
pytest tests
[...]
Using --random-order-bucket=module
Using --random-order-seed=573663
```

So that if the given particular sequence fails, you can reproduce it by adding that exact seed, e.g.:

```
pytest --random-order-seed=573663
[...]
Using --random-order-bucket=module
Using --random-order-seed=573663
```

It will only reproduce the exact order if you use the exact same list of tests (or no list at all). Once you start to manually narrowing down the list you can no longer rely on the seed, but have to list them manually in the exact order they failed and tell pytest to not randomize them instead using `--random-order-bucket=none`, e.g.:

```
pytest --random-order-bucket=none tests/test_a.py tests/test_c.py tests/test_b.py
```

To disable the shuffling for all tests:

```
pytest --random-order-bucket=none
```

By default `--random-order-bucket=module` is implied, which will shuffle the files on the module levels. It can also shuffle on `class`, `package`, `global` and `none` levels. For the complete details please see its documentation.

Another randomization alternative is: `pytest-randomly`. This module has a very similar functionality/interface, but it doesn't have the bucket modes available in `pytest-random-order`. It has the same problem of imposing itself once installed.

Look and feel variations

pytest-sugar

pytest-sugar is a plugin that improves the look-n-feel, adds a progressbar, and show tests that fail and the assert instantly. It gets activated automatically upon installation.

```
pip install pytest-sugar
```

To run tests without it, run:

```
pytest -p no:sugar
```

or uninstall it.

Report each sub-test name and its progress

For a single or a group of tests via `pytest` (after `pip install pytest-pspec`):

```
pytest --pspec tests/test_optimization.py
```

Instantly shows failed tests

pytest-instafail shows failures and errors instantly instead of waiting until the end of test session.

```
pip install pytest-instafail
```

```
pytest --instafail
```

To GPU or not to GPU

On a GPU-enabled setup, to test in CPU-only mode add `CUDA_VISIBLE_DEVICES=""`:

```
CUDA_VISIBLE_DEVICES="" pytest tests/utils/test_logging.py
```

or if you have multiple gpus, you can specify which one is to be used by `pytest`. For example, to use only the second gpu if you have gpus `0` and `1`, you can run:

```
CUDA_VISIBLE_DEVICES="1" pytest tests/utils/test_logging.py
```

This is handy when you want to run different tasks on different GPUs.

Some tests must be run on CPU-only, others on either CPU or GPU or TPU, yet others on multiple-GPUs. The following skip decorators are used to set the requirements of tests CPU/GPU/TPU-wise:

- `require_torch` - this test will run only under torch
- `require_torch_gpu` - as `require_torch` plus requires at least 1 GPU
- `require_torch_multi_gpu` - as `require_torch` plus requires at least 2 GPUs
- `require_torch_non_multi_gpu` - as `require_torch` plus requires 0 or 1 GPUs
- `require_torch_up_to_2_gpus` - as `require_torch` plus requires 0 or 1 or 2 GPUs
- `require_torch_tpu` - as `require_torch` plus requires at least 1 TPU

Let's depict the GPU requirements in the following table:

n gpus	decorator
`>= 0`	`@require_torch`
`>= 1`	`@require_torch_gpu`
`>= 2`	`@require_torch_multi_gpu`
`< 2`	`@require_torch_non_multi_gpu`
`< 3`	`@require_torch_up_to_2_gpus`

For example, here is a test that must be run only when there are 2 or more GPUs available and pytorch is installed:

```
from testing_utils import require_torch_multi_gpu

@require_torch_multi_gpu
def test_example_with_multi_gpu():
```

These decorators can be stacked:

```
from testing_utils import require_torch_gpu

@require_torch_gpu
@some_other_decorator
def test_example_slow_on_gpu():
```

Some decorators like `@parametrized` rewrite test names, therefore `@require_*` skip decorators have to be listed last for them to work correctly. Here is an example of the correct usage:

```
from testing_utils import require_torch_multi_gpu
from parameterized import parameterized

@parameterized.expand(...)
@require_torch_multi_gpu
def test_integration_foo():
```

This order problem doesn't exist with `@pytest.mark.parametrize`, you can put it first or last and it will still work. But it only works with non-unittests.

Inside tests:

- How many GPUs are available:

```
from testing_utils import get_gpu_count

n_gpu = get_gpu_count()
```

Distributed training

`pytest` can't deal with distributed training directly. If this is attempted - the sub-processes don't do the right thing and end up thinking they are `pytest` and start running the test suite in loops. It works, however, if one spawns a normal process that then spawns off multiple workers and manages the IO pipes.

Here are some tests that use it:

- test_trainer_distributed.py
- test_deepspeed.py

To jump right into the execution point, search for the `execute_subprocess_async` call in those tests, which you will find inside testing_utils.py.

You will need at least 2 GPUs to see these tests in action:

```
CUDA_VISIBLE_DEVICES=0,1 RUN_SLOW=1 pytest -sv tests/test_trainer_distributed.py
```

(`RUN_SLOW` is a special decorator used by HF Transformers to normally skip heavy tests)

Output capture

During test execution any output sent to `stdout` and `stderr` is captured. If a test or a setup method fails, its according captured output will usually be shown along with the failure traceback.

To disable output capturing and to get the `stdout` and `stderr` normally, use `-s` or `--capture=no`:

```
pytest -s tests/utils/test_logging.py
```

To send test results to JUnit format output:

```
py.test tests --junitxml=result.xml
```

Color control

To have no color (e.g., yellow on white background is not readable):

```
pytest --color=no tests/utils/test_logging.py
```

Sending test report to online pastebin service

As of pytest 9.1.0 this flag lives in the separate `pytest-pastebin` plugin (`pip install pytest-pastebin`).

Creating a URL for each test failure:

```
pytest --pastebin=failed tests/utils/test_logging.py
```

This will submit test run information to a remote Paste service and provide a URL for each failure. You may select tests as usual or add for example -x if you only want to send one particular failure.

Creating a URL for a whole test session log:

```
pytest --pastebin=all tests/utils/test_logging.py
```

Writing tests

Most of the time if combining `pytest` and `unittest` in the same test suite works just fine. You can read here which features are supported when doing that , but the important thing to remember is that most `pytest` fixtures don't work. Neither parametrization, but we use the module `parameterized` that works in a similar way.

Parametrization

Often, there is a need to run the same test multiple times, but with different arguments. It could be done from within the test, but then there is no way of running that test for just one set of arguments.

```
# test_this1.py
import unittest
from parameterized import parameterized


class TestMathUnitTest(unittest.TestCase):
    @parameterized.expand(
        [
            ("negative", -1.5, -2.0),
            ("integer", 1, 1.0),
            ("large fraction", 1.6, 1),
        ]
    )
    def test_floor(self, name, input, expected):
        assert_equal(math.floor(input), expected)
```

Now, by default this test will be run 3 times, each time with the last 3 arguments of `test_floor` being assigned the corresponding arguments in the parameter list.

And you could run just the `negative` and `integer` sets of params with:

```
pytest -k "negative and integer" tests/test_mytest.py
```

or all but `negative` sub-tests, with:

```
pytest -k "not negative" tests/test_mytest.py
```

Besides using the `-k` filter that was just mentioned, you can find out the exact name of each sub-test and run any or all of them using their exact names.

```
pytest test_this1.py --collect-only -q
```

and it will list:

```
test_this1.py::TestMathUnitTest::test_floor_0_negative
test_this1.py::TestMathUnitTest::test_floor_1_integer
test_this1.py::TestMathUnitTest::test_floor_2_large_fraction
```

So now you can run just 2 specific sub-tests:

```
pytest test_this1.py::TestMathUnitTest::test_floor_0_negative
test_this1.py::TestMathUnitTest::test_floor_1_integer
```

The module parameterized works for both: `unittests` and `pytest` tests.

If, however, the test is not a `unittest`, you may use `pytest.mark.parametrize`.

Here is the same example, this time using `pytest`'s `parametrize` marker:

```
# test_this2.py
import pytest


@pytest.mark.parametrize(
    "name, input, expected",
    [
        ("negative", -1.5, -2.0),
        ("integer", 1, 1.0),
        ("large fraction", 1.6, 1),
    ],
)
def test_floor(name, input, expected):
    assert_equal(math.floor(input), expected)
```

Same as with `parameterized`, with `pytest.mark.parametrize` you can have a fine control over which sub-tests are run, if the `-k` filter doesn't do the job. Except, this parametrization function creates a slightly different set of names for the sub-tests. Here is what they look like:

```
pytest test_this2.py --collect-only -q
```

and it will list:

```
test_this2.py::test_floor[integer-1-1.0]
test_this2.py::test_floor[negative--1.5--2.0]
test_this2.py::test_floor[large fraction-1.6-1]
```

So now you can run just the specific test:

```
pytest test_this2.py::test_floor[negative--1.5--2.0] test_this2.py::test_floor[integer-1-1.0]
```

as in the previous example.

Files and directories

In tests often we need to know where things are relative to the current test file, and it's not trivial since the test could be invoked from more than one directory or could reside in sub-directories with different depths. A helper class `testing_utils.TestCasePlus` solves this problem by sorting out all the basic paths and provides easy accessors to them:

- `pathlib` objects (all fully resolved):
    - `test_file_path` - the current test file path, i.e. `__file__`
    - `test_file_dir` - the directory containing the current test file
    - `tests_dir` - the directory of the `tests` test suite
    - `examples_dir` - the directory of the `examples` test suite
    - `repo_root_dir` - the directory of the repository
    - `src_dir` - the directory of `src` (i.e. where the `transformers` sub-dir resides)
- stringified paths -- same as above but these return paths as strings, rather than `pathlib` objects:
    - `test_file_path_str`
    - `test_file_dir_str`
    - `tests_dir_str`
    - `examples_dir_str`
    - `repo_root_dir_str`
    - `src_dir_str`

To start using those all you need is to make sure that the test resides in a subclass of `testing_utils.TestCasePlus`. For example:

```
from testing_utils import TestCasePlus


class PathExampleTest(TestCasePlus):
    def test_something_involving_local_locations(self):
        data_dir = self.tests_dir / "fixtures/tests_samples/wmt_en_ro"
```

If you don't need to manipulate paths via `pathlib` or you just need a path as a string, you can always invoked `str()` on the `pathlib` object or use the accessors ending with `_str`. For example:

```
from testing_utils import TestCasePlus


class PathExampleTest(TestCasePlus):
    def test_something_involving_stringified_locations(self):
        examples_dir = self.examples_dir_str
```

Temporary files and directories

Using unique temporary files and directories are essential for parallel test running, so that the tests won't overwrite each other's data. Also we want to get the temporary files and directories removed at the end of each test that created them. Therefore, using packages like `tempfile`, which address these needs is essential.

However, when debugging tests, you need to be able to see what goes into the temporary file or directory and you want to know it's exact path and not having it randomized on every test re-run.

A helper class `testing_utils.TestCasePlus` is best used for such purposes. It's a sub-class of `unittest.TestCase`, so we can easily inherit from it in the test modules.

Here is an example of its usage:

```
from testing_utils import TestCasePlus


class ExamplesTests(TestCasePlus):
    def test_whatever(self):
        tmp_dir = self.get_auto_remove_tmp_dir()
```

This code creates a unique temporary directory, and sets `tmp_dir` to its location.

- Create a unique temporary dir:

```
def test_whatever(self):
    tmp_dir = self.get_auto_remove_tmp_dir()
```

`tmp_dir` will contain the path to the created temporary dir. It will be automatically removed at the end of the test.

- Create a temporary dir of my choice, ensure it's empty before the test starts and don't empty it after the test.

```
def test_whatever(self):
    tmp_dir = self.get_auto_remove_tmp_dir("./xxx")
```

This is useful for debug when you want to monitor a specific directory and want to make sure the previous tests didn't leave any data in there.

- You can override the default behavior by directly overriding the `before` and `after` args, leading to one of the following behaviors:
    - `before=True`: the temporary dir will always be cleared at the beginning of the test.
    - `before=False`: if the temporary dir already existed, any existing files will remain there.
    - `after=True`: the temporary dir will always be deleted at the end of the test.
    - `after=False`: the temporary dir will always be left intact at the end of the test.

footnote: In order to run the equivalent of `rm -r` safely, only subdirs of the project repository checkout are allowed if an explicit `tmp_dir` is used, so that by mistake no `/tmp` or similar important part of the filesystem will get nuked. i.e. please always pass paths that start with `./`.

footnote: Each test can register multiple temporary directories and they all will get auto-removed, unless requested otherwise.

Temporary sys.path override

If you need to temporary override `sys.path` to import from another test for example, you can use the `ExtendSysPath` context manager. Example:

```
import os
from testing_utils import ExtendSysPath

bindir = os.path.abspath(os.path.dirname(__file__))
with ExtendSysPath(f"{bindir}/.."):
    from test_trainer import TrainerIntegrationCommon  # noqa
```

Skipping tests

This is useful when a bug is found and a new test is written, yet the bug is not fixed yet. In order to be able to commit it to the main repository we need make sure it's skipped during `make test`.

Methods:

- A **skip** means that you expect your test to pass only if some conditions are met, otherwise pytest should skip running the test altogether. Common examples are skipping windows-only tests on non-windows platforms, or skipping tests that depend on an external resource which is not available at the moment (for example a database).
- A **xfail** means that you expect a test to fail for some reason. A common example is a test for a feature not yet implemented, or a bug not yet fixed. When a test passes despite being expected to fail (marked with `pytest.mark.xfail`), it's an xpass and will be reported in the test summary.

One of the important differences between the two is that `skip` doesn't run the test, and `xfail` does. So if the code that's buggy causes some bad state that will affect other tests, do not use `xfail`.

Implementation

- Here is how to skip whole test unconditionally:

```
@unittest.skip("this bug needs to be fixed")
def test_feature_x():
```

or via pytest:

```
@pytest.mark.skip(reason="this bug needs to be fixed")
```

or the `xfail` way:

```
@pytest.mark.xfail
def test_feature_x():
```

Here's how to skip a test based on internal checks within the test:

```
def test_feature_x():
    if not has_something():
        pytest.skip("unsupported configuration")
```

or the whole module (via an autouse fixture — `pytest.config` was removed in pytest 4.0):

```
import pytest

@pytest.fixture(scope="module", autouse=True)
def _require_custom_flag(pytestconfig):
    if not pytestconfig.getoption("--custom-flag", default=False):
        pytest.skip("--custom-flag is missing, skipping tests")
```

or the `xfail` way:

```
def test_feature_x():
    pytest.xfail("expected to fail until bug XYZ is fixed")
```

- Here is how to skip all tests in a module if some import is missing:

```
docutils = pytest.importorskip("docutils", minversion="0.3")
```

- Skip a test based on a condition:

```
@pytest.mark.skipif(sys.version_info < (3,6), reason="requires python3.6 or higher")
def test_feature_x():
```

or:

```
@unittest.skipIf(torch_device == "cpu", "Can't do half precision")
def test_feature_x():
```

or skip the whole module:

```
@pytest.mark.skipif(sys.platform == 'win32', reason="does not run on windows")
class TestClass():
    def test_feature_x(self):
```

More details, example and ways are here.

Capturing outputs

Capturing the stdout/stderr output

In order to test functions that write to `stdout` and/or `stderr`, the test can access those streams using the `pytest`'s capsys system. Here is how this is accomplished:

```
import sys


def print_to_stdout(s):
    print(s)


def print_to_stderr(s):
    sys.stderr.write(s)


def test_result_and_stdout(capsys):
    msg = "Hello"
    print_to_stdout(msg)
    print_to_stderr(msg)
    out, err = capsys.readouterr()  # consume the captured output streams
    # optional: if you want to replay the consumed streams:
    sys.stdout.write(out)
    sys.stderr.write(err)
    # test:
    assert msg in out
    assert msg in err
```

And, of course, most of the time, `stderr` will come as a part of an exception, so try/except has to be used in such a case:

```
def raise_exception(msg):
    raise ValueError(msg)


def test_something_exception():
    msg = "Not a good value"
    error = ""
    try:
```

```
        raise_exception(msg)
    except Exception as e:
        error = str(e)
        assert msg in error, f"{msg} is in the exception:\n{error}"
```

Another approach to capturing stdout is via `contextlib.redirect_stdout`:

```
from io import StringIO
from contextlib import redirect_stdout


def print_to_stdout(s):
    print(s)


def test_result_and_stdout():
    msg = "Hello"
    buffer = StringIO()
    with redirect_stdout(buffer):
        print_to_stdout(msg)
    out = buffer.getvalue()
    # optional: if you want to replay the consumed streams:
    sys.stdout.write(out)
    # test:
    assert msg in out
```

An important potential issue with capturing stdout is that it may contain `\r` characters that in normal `print` reset everything that has been printed so far. There is no problem with `pytest`, but with `pytest -s` these characters get included in the buffer, so to be able to have the test run with and without `-s`, you have to make an extra cleanup to the captured output, using `re.sub(r'~.*\r', '', buf, 0, re.M)`.

But, then we have a helper context manager wrapper to automatically take care of it all, regardless of whether it has some `\r`'s in it or not, so it's a simple:

```
from testing_utils import CaptureStdout

with CaptureStdout() as cs:
    function_that_writes_to_stdout()
print(cs.out)
```

Here is a full test example:

```
from testing_utils import CaptureStdout

msg = "Secret message\r"
final = "Hello World"
with CaptureStdout() as cs:
```

```
    print(msg + final)
assert cs.out == final + "\n", f"captured: {cs.out}, expecting {final}"
```

If you'd like to capture `stderr` use the `CaptureStderr` class instead:

```
from testing_utils import CaptureStderr

with CaptureStderr() as cs:
    function_that_writes_to_stderr()
print(cs.err)
```

If you need to capture both streams at once, use the parent `CaptureStd` class:

```
from testing_utils import CaptureStd

with CaptureStd() as cs:
    function_that_writes_to_stdout_and_stderr()
print(cs.err, cs.out)
```

Also, to aid debugging test issues, by default these context managers automatically replay the captured streams on exit from the context.

Capturing logger stream

If you need to validate the output of a logger, you can use `CaptureLogger`:

```
from transformers import logging
from testing_utils import CaptureLogger

msg = "Testing 1, 2, 3"
logging.set_verbosity_info()
logger = logging.get_logger("transformers.models.bart.tokenization_bart")
with CaptureLogger(logger) as cl:
    logger.info(msg)
assert cl.out, msg + "\n"
```

Testing with environment variables

If you want to test the impact of environment variables for a specific test you can use a helper decorator `transformers.testing_utils.mockenv`

```
from testing_utils import mockenv


class HfArgumentParserTest(unittest.TestCase):
    @mockenv(TRANSFORMERS_VERBOSITY="error")
```

```
    def test_env_override(self):
        env_level_str = os.getenv("TRANSFORMERS_VERBOSITY", None)
```

At times an external program needs to be called, which requires setting `PYTHONPATH` in `os.environ` to include multiple local paths. A helper class `testing_utils.TestCasePlus` comes to help:

```
from testing_utils import TestCasePlus


class EnvExampleTest(TestCasePlus):
    def test_external_prog(self):
        env = self.get_env()
        # now call the external program, passing `env` to it
```

Depending on whether the test file was under the `tests` test suite or `examples` it'll correctly set up `env[PYTHONPATH]` to include one of these two directories, and also the `src` directory to ensure the testing is done against the current repo, and finally with whatever `env[PYTHONPATH]` was already set to before the test was called if anything.

This helper method creates a copy of the `os.environ` object, so the original remains intact.

Getting reproducible results

When a test needs identical results across runs, fix the RNGs and the GPU determinism knobs - seeding alone is not enough on CUDA. The full recipe lives in Achieve determinism in randomness based software; call that chapter's `enforce_reproducibility(seed)` (or the equivalent settings it documents) from the test rather than maintaining a second, incomplete copy here.

Debugging tests

To start a debugger at the point of the warning, do this:

```
pytest tests/utils/test_logging.py -W error::UserWarning --pdb
```

A massive hack to create multiple pytest reports

Here is a massive `pytest` patching that I have done many years ago to aid with understanding CI reports better.

To activate it add to `tests/conftest.py` (or create it if you haven't already):

```
import pytest

def pytest_addoption(parser):
    from testing_utils import pytest_addoption_shared

    pytest_addoption_shared(parser)


def pytest_terminal_summary(terminalreporter):
```

```
    from testing_utils import pytest_terminal_summary_main

    make_reports = terminalreporter.config.getoption("--make-reports")
    if make_reports:
        pytest_terminal_summary_main(terminalreporter, id=make_reports)
```

and then when you run the test suite, add `--make-reports=mytests` like so:

```
pytest --make-reports=mytests tests
```

and it'll create 8 separate reports:

```
$ ls -1 reports/mytests/
durations.txt
errors.txt
failures_line.txt
failures_long.txt
failures_short.txt
stats.txt
summary_short.txt
warnings.txt
```

so now instead of having only a single output from `pytest` with everything together, you can now have each type of report saved into each own file.

This feature is most useful on CI, which makes it much easier to both introspect problems and also view and download individual reports.

Using a different value to `--make-reports=` for different groups of tests can have each group saved separately rather than clobbering each other.

All this functionality was already inside `pytest` but there was no way to extract it easily so I added the monkey-patching overrides testing_utils.py. Well, I did ask if I can contribute this as a feature to `pytest` but my proposal wasn't welcome.

Resources

Similar online guides

- Boris Dayma wrote A Recipe for Training Large Models
- The HuggingFace team published The Ultra-Scale Playbook: Training LLMs on GPU Clusters.
- Jacob Austin maintains How to Scale Your Model: A Systems View of LLMs on TPUs
- The Harvard Edge group publishes Machine Learning Systems (CS249r textbook; site: mlsysbook.ai).

Useful compilations

- @StellaAthena created the Common LLM Settings spreadsheet which can be a super-useful resource when you're about to embark on a new LLM training - as it tells you how many known LLM trainings were created.
- A few years back I started compiling information on which dtype the models were trained in - it only contains a handful of models but if you're doing a research on dtypes it can still be useful. I was using this information to try and write a model pretraining dtype auto-detection and here is a related float16 vs bfloat16 numerical properties comparison.

Publicly available training LLM/VLM logbooks

Logbooks and chronicles of training LLM/VLM are one of the best sources to learn from about dealing with training instabilities and choosing good hyper parameters.

If you know of a public LLM/VLM training logbook that is not on this list please kindly let me know or add it via a PR. Thank you!

The listing is in no particular order other than being grouped by the year.

2021

- BigScience pre-BLOOM 108B training experiments (2021): chronicles | the full spec and discussions (backup: 1 | 2)

2022

- BigScience BLOOM-176B (2022): chronicles-prequel | chronicles | the full spec and discussions (backup: 1 | 2 | 3)
- Meta OPT-175B (2022): logbook | Video (backup: 1)
- THUDM GLM-130B (2022): en logbook | Mandarin version (backup: 1 | 2)

2023

- HuggingFace IDEFICS-80B multimodal (Flamingo repro) (2023): Learning log | Training Chronicles (backup: 1 | 2)
- BloombergGPT 50B LLM - section C in BloombergGPT: A Large Language Model for Finance

2024

- MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs - the paper covers various training issues and their resolution - albeit on models that are proprietary yet just as instructional/useful.
- Imbue's From bare metal to a 70B model: infrastructure set-up and scripts very detailed technical post covers many training-related issues that they had to overcome while training a proprietary 70B-param model.

Hardware setup logbooks

- Imbue published a detailed log of how they have set up a 512-node IB-fat-tree cluster and made it to work: From bare metal to a 70B model: infrastructure set-up and scripts, they also open-sourced the cluster tooling they created in the process.
- SemiAnalysis published a great detailed writeup about what it takes to set up a Neocloud cluster.

Contributors

Multiple contributors kindly helped to improve these ever improving and expanding notes.

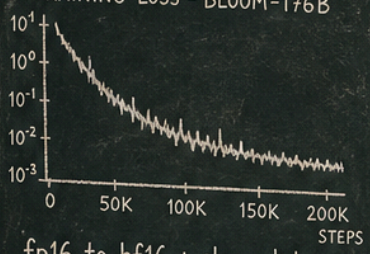
1. Some of them did it via PRs, and are thus listed automatically here
2. Others did it via various other ways so I'm listing them explicitly here:

- Adam Moody
- Alex Rogozhnikov
- Bowei Liu
- Darrick Horton
- Elio VP
- Garrett Goon
- Horace He
- Ivan Yashchuk
- Jack Dent
- Jeff Rasley
- Jon Stevens
- Jordan Nanos
- Mark Saroufim
- Olatunji Ruwase
- Oren Leung
- Quentin Anthony
- Ross Wightman
- Samyam Rajbhandari
- Shikib Mehri
- Siddharth Singh
- Stéphane Requena
- Zhiqi Tao

If you contributed to this text and for some reason you're not on one of these 2 lists - let's fix it by adding your name with a github or similar link here.

Lessons Learned from Training LLMs (a course)

TRAINING LOSS - BLOOM-176B



fp16 to bf16 + clean data + emb layer-norm = no more divergence

STD Init

$$\text{sqrt}\left(\frac{1}{\text{NHIDDEN} * 3}\right) = 0.00482$$

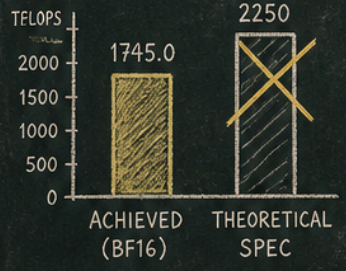
for NHIDDEN = 14336

one of the reasons BLOOM-176B never diverged



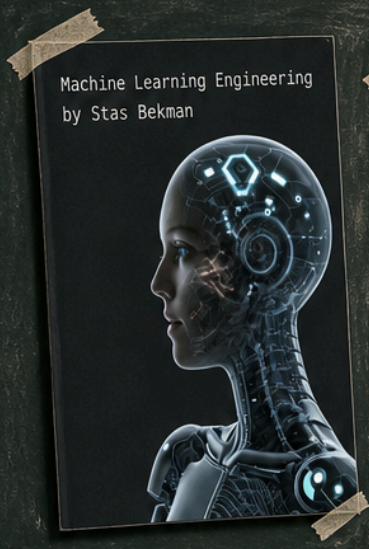
JeanZay HPC 2021: 2.3TB checkpoint saved in 40 sec across 384 processes (GPFS over NVMe)

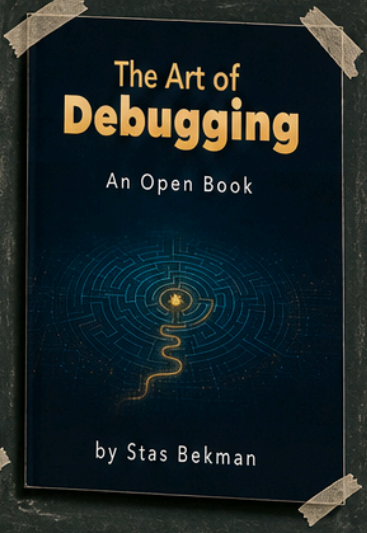
MAMF



77.6% efficiency - spec sheets lie

```
$
CUDA_LAUNCH_BLOCKING=1
- zero slowdown!
```

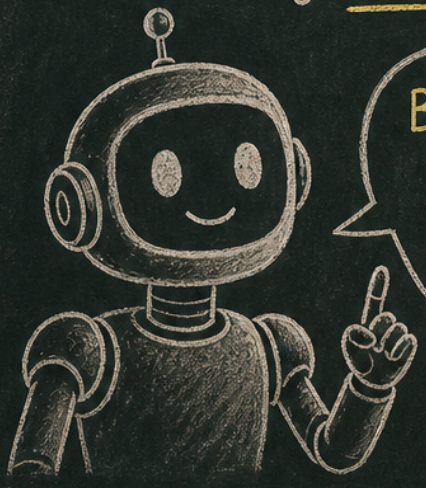




Machine Learning Engineering

Lessons Learned from Training LLMs

A course by Stas Bekman



STORAGE - NETWORK - PERFORMANCE - STABILITY - DEBUGGING

FROM TRAINING BLOOM-176B, IDEFICS-80B & PRODUCTION RAG/RL SYSTEMS

A curated collection of real-world "battle stories" and their resolutions, based on two open books by Stas Bekman:

- Machine Learning Engineering (MLE)
- The Art of Debugging (AoD)

Each item below is a hard-won lesson from actually training and operating large language/image models at scale, along with the hardware, storage, network, and debugging surprises that came with them:

- BLOOM-176B (2022)
- IDEFICS-80B (2023)
- Production RAG/RL systems at Contextual.AI (2024)
- Various Arctic Training open source work at Snowflake (2025-2026)

Each lesson's bold headline is the entry point, and the supporting facts underneath it are broken out as scannable bullets instead of a flowing paragraph, so you can skim the "what/why/how" quickly and still drop into the link for the full story.

This is meant to be read as a course: read the headlines to identify areas of interest, skim the sub-bullets for the reasoning, and if more details are wanted, follow the read-more links to dive into the details, scripts, and benchmarks.

You can take it as a self-guided course, or teach it to others yourself!

Chapters

1. Storage and File Systems
2. Network and Interconnects
3. Compute, Accelerators and Memory
4. Training Performance and Throughput
5. DataLoaders
6. Fault Tolerance and Reliability
7. Training Stability
8. Cloud Providers and Procurement
9. Debugging Methodology

1. Storage and File Systems

- 1.1. **Fast parallel storage is a game changer for fast model checkpointing.**
    - Distributed parallel file systems like GPFS and WekaIO let hundreds to thousands of clients hit the same storage at once without creating hotspots - the difference this makes is dramatic.
    - **Case in point:** at JeanZay HPC (2021), a 2.3TB checkpoint was written in parallel across 384 processes in just 40 seconds, because the cluster ran GPFS over NVMe drives.
    - The gap between "GPFS over NVMe" and "a generic cloud NFS mount" can be many multiples in wall-clock time.
    - **Takeaway:** benchmark your specific file system for your specific workload to know the overheads.

  📖 Which file system to choose

- 1.2. **Slow shared storage silently kills developer productivity, not just training throughput.**
    - On GCP's Zonal FileStore over NFS, a plain `python -c "import torch"` took 20 seconds uncached (versus ~2 seconds once cached), and installing a conda environment with a handful of prebuilt packages could take 20-30 minutes.
    - This kind of slowdown doesn't show up in a training-throughput benchmark at all, but it grinds down every developer, every day, multiplied across the whole team.
    - **The fix:** run a "perception benchmark" (timed `import torch`, `conda create`, `git status`,

`pytest --collect-only`) against any storage solution before committing to it, comparing it to a local NVMe baseline.
    - If a proposed shared filesystem is only 2x faster than what you have, it's probably not worth switching - aim for something like 5-10x closer to local NVMe speeds before it stops being a daily source of frustration.
    - Start using `uv` - 5-50x faster than `pip install`.

  📖 Which file system to choose

- 1.3. **Python's tens-of-thousands of tiny files can exhaust inodes long before you run out of actual disk space.**
    - If a filesystem partition uses a large block size (e.g. 16MiB, common for GPFS partitions optimized for big files) but your files average 16KiB, you waste up to 1,024x the space per file - and, more importantly, chew through the inode budget instead of the byte budget.
    - This is exactly what happened at JeanZay HPC: conda installs tens of thousands of tiny 4KiB-ish files per environment, so the team kept running out of inodes on normal GPFS partitions and had to request a dedicated partition just for conda environments.
    - **The fix:** keep at least two differently-tuned partitions - a small (4-8KiB) block size for millions of tiny files (venvs, package caches), and a large (2-16MiB) block size for a few huge files (checkpoints, datasets). Modern GPFS sub-blocks can also help by packing tiny files together.
    - Always check `df -i` alongside `df -h`, since a partition can report plenty of free space while being completely out of usable inodes.

  📖 File Block size

- 1.4. **Dataloading directly from cloud storage during training is painful and should be avoided if at all possible.**
    - During IDEFICS-80B training there wasn't enough local disk space for the multi-TB multimodal dataset, so the team had to stream from cloud storage - and it took many weeks of effort trying to make that solution robust, without ever quite getting there.
    - **The core problem:** the streaming Dataloader at that time didn't preserve the `DataSampler`'s RNG state across resumes, so every crash-and-resume cycle either skipped or repeated data - the training never got a single clean epoch of genuinely unique data.
    - **Lesson:** prefer having enough local disk for dataloading, and only offload checkpoints to the cloud asynchronously (via crontab or a SLURM job), always keeping the last couple of checkpoints locally for a fast resume.
    - If you must stream from the cloud, make sure your specific solution can resume exactly - without losing or repeating data, and without needing an enormous number of local DataLoader worker processes.

  📖 Local storage beats cloud storage

2. Network and Interconnects

- 2.1. **Always double check whether a bandwidth number is unidirectional or bidirectional (duplex) before comparing it to a measurement.**
    - Most benchmarking tools report unidirectional bandwidth, but vendor marketing nearly always prefers the duplex number, since it's roughly 2x bigger and looks more impressive.
    - **Rule of thumb:** if your measured bandwidth comes out to around 40% of the advertised spec, the advertised number was probably duplex - halve it, and your real efficiency is a common ~80%.
    - **Example:** on an A100 node advertised at 600GBps intra-node, a measured 235GBps looked

like a disappointing 40% efficiency, until it was compared against the correct 300GBps unidirectional spec (~80%).
    - The same pattern repeats on H200 today: NVLink is advertised at 900GBps (duplex), but an all-reduce benchmark reaching ~376GBps checks out nicely against the 450GBps unidirectional spec.

📖 Unidirectional vs Bidirectional (Duplex)

- 2.2. **A slow inter-node network can force your entire training framework choice, not just your throughput ceiling.**
    - At JeanZay HPC, the Omni-Path interconnect was stuck at only 135Gbps.
    - That single fact forced the BLOOM-176B team onto Megatron-DeepSpeed (tensor + pipeline parallelism, which sends far less data over the wire) instead of the much simpler DeepSpeed ZeRO (which needs much higher bandwidth to shard optimizer states and gradients).
    - This kind of decision is nearly impossible to walk back once a multi-month training run is underway - network speed needs to be known and factored into your parallelism strategy at the very start of the project.
    - If you're stuck with a slow inter-node network, favor strategies that only exchange activations (pipeline/tensor parallelism) over ones that exchange full gradients and optimizer state (ZeRO-DP/FSDP).

📖 Omni-Path

- 2.3. **Sometimes an all-reduce benchmark reports a number that seems to exceed the physical wire's theoretical maximum, and that's not a measurement bug - it's SHARP.**
    - On an H100 node, an intra-node NVLink 4.0 all-reduce reported 480GBps for a 4GiB payload - even though the unidirectional spec is "only" 450GBps.
    - **The explanation:** NVIDIA's SHARP protocol lets NCCL perform the reduction directly on network switch hardware. Once NCCL detects InfiniBand/NVSwitch SHARP support, it switches to the `NVLS` algorithm, which needs only `N+1` sends instead of the usual `2N`.
    - This also means the standard `busbw` calculation (which assumes the `2N` model) under-reports true efficiency once `NVLS`/SHARP kicks in - a number that looks "too good" is a smarter algorithm at work, not an instrumentation error.
    - The speedup needs more than 4 GPUs in the collective, and `NCCL_NVLS_ENABLE` env var not disabled.
    - SHARP applies to `all-reduce` only. A full-node `all-gather` or `reduce-scatter` runs the ring path and lands at the ordinary ~80% of the unidirectional spec.

📖 SHARP

- 2.4. **A shared inter-node network makes performance optimization nearly impossible, because you can't tell what changed - your code or your noisy neighbors.**
    - At JeanZay HPC, before BLOOM-176B, the exact same configuration produced meaningfully different throughput on every single run, simply because other users' jobs were sharing the same fabric at unpredictable intensity.
    - This is a dead end for serious tuning work: you can't attribute a throughput change to your own tweak if the ambient noise floor itself shifts by double digits between runs.
    - **The fix:** the team was granted an exclusive, isolated SLURM partition on the (then brand new) A100s just before the BLOOM-176B launch, at which point optimization work finally became tractable.
    - If you're doing serious pre-training performance work, insist on a dedicated, isolated network partition for the benchmarking phase.

📖 Shared internode network + Crucial reproducibility requirements

3. Compute, Accelerators and Memory

- 3.1. **Coherent (unified) memory hardware has genuinely unpredictable behavior, and the fix is a driver-level mode switch, not a code change.**
    - On NVIDIA's coherent-memory platforms (DGX Station, DGX Spark, GH200, GB300), Linux can transparently borrow from GPU HBM when it runs out of CPU RAM - but not the reverse - and `nvidia-smi` may not even report this borrowing.
    - While tuning max sequence length for Qwen3-32B post-training on a DGX Station (optimizer states offloaded to CPU RAM), the team hit maddening, seemingly random alternation between CPU-OOM and GPU-OOM, depending on whatever background daemons happened to be running.
    - **The fix** was not a code change at all, but a system setting: switching from the default NUMA coherent memory mode to CDMM ("driver") mode via `/etc/modprobe.d/nvidia-openrm.conf` and a reboot, after which CPU and GPU memory stopped silently cross-borrowing.
    - If you see unexplained, seemingly non-deterministic OOMs on coherent-memory NVIDIA hardware, check the `CoherentGPUMemoryMode` setting before debugging your training script.

  📖 Overcoming the coherent memory uncertain behavior

- 3.2. **Cryptic async CUDA errors sometimes need the "debug" flag turned on permanently, and that flag can be free.**
    - CUDA is async by default, so by the time an error manifests, the program's context has already moved on - producing tracebacks that point nowhere useful, or NCCL messages that outright refuse to generate one.
    - `CUDA_LAUNCH_BLOCKING=1` forces synchronous execution, normally at a real performance cost, but it also means you finally get an accurate, in-context traceback when something breaks.
    - BLOOM-176B ran its *entire* training under this flag (because training would otherwise hang for an undiagnosed reason), and surprisingly saw zero measurable throughput impact.
    - **Lesson:** don't assume a debug flag is automatically a performance tax without measuring it for your specific case - it might also be the fix.

  📖 Dealing with Async CUDA bugs

- 3.3. **Memory profilers catch real leaks that are otherwise nearly invisible in code review.**
    - While implementing a tricky custom `TiledMLP` `torch.autograd.Function`, a leak was hiding in the implementation and only became obvious once rendered visually via PyTorch's memory profiler (`torch.cuda.memory._record_memory_history()` + `_dump_snapshot()`, viewed at pytorch.org/memory_viz).
    - Most allocations show up as bars that start and get freed. The leaked tensors showed up as continuous bars that started at one layer's forward pass and never ended, repeating on every subsequent layer.
    - You can click any bar to get the Python-level traceback to the exact allocating line, turning "there's a leak somewhere" into "line X in file Y" in minutes.
    - Beyond leak-hunting, the same tool is great for visually comparing how differently two implementations of "the same" algorithm use GPU memory over time (e.g. normal vs. CPU-offloaded activation checkpointing).

  📖 PyTorch memory profiler

- 3.4. **Strategic, precisely-placed memory allocation tracing finds framework bugs too, not just bugs in your own code.**

    - Having explicit control over exactly *when* you sample GPU/CPU memory (a "sample now" call at strategic points, via a helper derived from DeepSpeed's `see_memory_usage`) can isolate a leak that a passive profiler might miss.
    - Using this technique, strategic tracing exposed a genuine leak inside PyTorch's own `all_gather_object` (since fixed, along with several other PyTorch-internal leaks found the same way).
    - **Nuance:** releasing a Python variable doesn't necessarily free the tensor immediately, since Python garbage collection runs on its own schedule. Call `gc.collect()` before sampling, or you'll see phantom "still allocated" numbers.
    - You must disable this tooling in production, as it will lead to bad performance.

  📖 Strategic memory allocation tracing

- 3.5. **Hyper-Threads can wreck network throughput on some hardware, and the only way to know is to actually benchmark both settings on your specific setup.**

    - On AWS p4 nodes, enabling Hyper-Threading (normally thought of as "free" extra CPU parallelism) dropped network throughput to roughly a quarter of its non-HT value.
    - Because of this specific, measured result, HT was deliberately kept disabled on that setup from then on, rather than left at the (often HT-enabled-by-default) SLURM configuration.
    - This is not universal in either direction - on some *other* setups the opposite is true, and disabling HT (`--hint=nomultithread`) is what degrades throughput.
    - **The only reliable path:** benchmark your actual workload's network throughput with HT on and off on your specific hardware, and pick whichever wins empirically.

  📖 To enable Hyper-Threads or not

4. Training Performance and Throughput

- 4.1. **Spec sheets provide unachievable numbers - always benchmark matmul on your own hardware.**

    - Theoretical peak TFLOPS assume perfectly-shaped matrices, no memory movement overhead, and boost clocks that hold indefinitely - none of which happens in real training.
    - **Example:** an NVIDIA B200 SXM may only reach ~1745.0TFLOPS (BF16) out of a 2250TFLOPS theoretical spec (~77.6% efficiency), and this number itself depends on the software stack version and exact matrix shapes.
    - This is why "Maximum Achievable Matmul FLOPS" (MAMF) exists as a metric distinct from the theoretical spec: it's obtained by brute-force searching matmul shapes on your actual hardware/software with `mamf-finder.py`, giving you the realistic ceiling to optimize against.
    - Once your measured training TFLOPS get close to your own MAMF number (not the theoretical peak), it's time to stop optimizing and start training.

  📖 Maximum Achievable and Sustainable Matmul FLOPS comparison table

- 4.2. **Throughput optimization is a journey with diminishing returns, not a fixed destination you arrive at instantly.**

    - During BLOOM-176B's tuning phase, the team started below 100TFLOPS and, over a few weeks, worked up to 150TFLOPS by launch - no single silver bullet, just accumulated smaller wins.
    - **Know when to stop:** once there was no more meaningful headroom relative to what similar setups achieved at the time (~155TFLOPS ceiling for 80GB A100s in 2022), the team deliberately stopped optimizing and began training.
    - Budget explicit calendar time for this tuning phase before the real launch - treating it as an afterthought means either wasting compute at low efficiency for months, or endlessly

delaying the launch trying to "perfect" the setup.

📖 TFLOPS as a performance metric

- 4.3. **Global batch size ramp-up matters far more than intuition suggests, especially with Pipeline Parallelism.**
    - For BLOOM-176B, starting too small was disastrous: GBS=16 measured only 8TFLOPS, because with Pipeline Parallelism a small GBS creates a large proportional "pipeline bubble" of idle GPU time.
    - The team instead started at GBS=192 (73TFLOPS) and ramped up by 16 every ~9.77M samples, eventually reaching GBS=2048 at 150TFLOPS.
    - **Why ramp at all:** early training is essentially random and doesn't benefit yet from large, highly-refined batches, so there's no point paying the pipeline-bubble cost of a small batch during that phase either.
    - If you use Pipeline Parallelism, benchmark your actual achieved TFLOPS at a few candidate starting GBS values before committing to a ramp schedule.

📖 Global Batch Size Ramp Up

5. DataLoaders

- 5.1. **Multiple DataLoaders in the same process can hang randomly, and the workaround may simply be to not use one of them.**
    - During BLOOM-176B, the training DataLoader ran fine with `num_workers=2`, but adding an evaluation DataLoader alongside it caused random hangs - a known class of long-unresolved PyTorch multi-worker-DataLoader bugs.
    - **The workaround:** set `num_workers=0` for the eval loader specifically (accepting slower, blocking IO since eval ran far less often), and eventually drop in-training eval entirely in favor of asynchronous lm-harness-style evaluation on saved checkpoints.
    - **Bonus:** removing the in-loop eval also sped up the main training loop, since it no longer shared resources with a second DataLoader at all.

📖 Asynchronous DataLoader

- 5.2. **Streaming DataLoaders can silently become your hard bottleneck if per-worker memory use is high and CPU RAM is limited.**
    - During IDEFICS-80B training, the streaming DataLoader for the multimodal dataset consumed huge, constantly-spiking memory per worker, and with only ~1TiB of CPU RAM per node the team couldn't spawn enough workers to keep the pipeline fed.
    - Unlike a GPU-memory bottleneck (which usually throws a clear error immediately), an under-provisioned DataLoader just quietly caps throughput below what the accelerators are capable of, without necessarily crashing anything.
    - There wasn't time to engineer a better solution, so the team finished the run accepting that DataLoader-bound ceiling.
    - If you use multimodal or otherwise memory-hungry streaming pipelines, budget CPU RAM headroom for DataLoader workers as carefully as you budget GPU memory for the model.

📖 Asynchronous DataLoader

6. Fault Tolerance and Reliability

- 6.1. **Beware of newly minted GPUs: a fresh batch of accelerators often has a 3-10% failure rate, and it's worse for the earliest units of a brand-new GPU generation.**

- This is a well-known enough pattern that burn-in practices exist to catch it before hardware reaches customers: OEMs typically run a 3-4 week burn-in, and cloud providers then run an additional 2-3 day acceptance test - though not every provider does this rigorously (or discloses whether they did).
- **Standard practical response:** plan for spare/redundant nodes (5-10% extra isn't unreasonable), and make sure your scheduler can automatically drain and replace bad nodes without manual intervention.
- Once you've spent weeks filtering good accelerators from bad ones, insist on a **fixed** allocation so those good nodes stay yours on reboot instead of going back into a shared pool.
- It's worth explicitly asking any cloud provider how long, and how rigorously, your specific nodes were burned in before you took possession - "we burned them in" without a duration is not a useful answer.

📖 Always plan to have more nodes than needed + Accelerators need to be burned in + Ensure you keep your good accelerators on reboot

- 6.2. **Checkpoint overhead scales inversely with your IO speed.**
    - BLOOM-176B's checkpoints were fast: 2.3TB in 40 seconds, every ~3 hours across a ~3-month run - about 720 checkpoints and only ~8 total hours of overhead, a mere ~0.37% tax on wall-clock time.
    - The same cadence, run on IO that's 5x slower (a normal gap between premium and standard cloud storage tiers), balloons to roughly 2% of total training time - a genuinely significant cost at scale.
    - **The math:** measure your actual save duration, multiply by how many times you intend to save, and compare that total to your expected training duration.
    - If you must offload checkpoints off-node to the cloud, always keep the two most recent checkpoints locally for a fast resume, since the very latest one might be corrupted from a mid-save crash.

📖 Frequent checkpoint saving

- 6.3. **Build a kill-switch, because SLURM has no concept of group job ownership and this becomes a real operational problem.**
    - In an HPC environment, if a colleague launches a long job (or a SLURM job array) and goes to sleep, you literally cannot stop or modify it without `sudo` access - SLURM's permission model is per-user, not per-team.
    - **BLOOM-176B's fix:** a file-poll kill-switch. The training loop checks for a specific file before each new iteration, and if found, saves a checkpoint and exits cleanly, letting *any* team member intervene.
    - An additional poll at the very start of `main()` means a long queued job array from a sleeping teammate can be "burned through" almost instantly, rather than each job running its full duration first.
    - Small amount of code, large amount of operational safety - worth building into any shared multi-user training setup from day one.

📖 Kill switch

- 6.4. **Watch the watchers: a scheduled job-status watchdog is what actually catches "the training silently stopped running and nobody noticed."**
    - During BLOOM-176B, a `slurm-status.py` script ran roughly hourly (via crontab-emulation on SLURM, since real `crontab` usually isn't available there) and checked whether the named job was running or at least queued.
    - If neither condition held, it emailed an alert and also piped the check into the main training log for a paper trail.

- This costs almost nothing to run, and it catches an entire category of expensive failures (hours or days of silently-idle compute) that would otherwise only surface when a human happened to check manually.
- **Principle:** don't just build the training loop - build the thing that watches the training loop.

📖 Is-job-running watchdog

- 6.5. **Monitor disk usage (and inode usage) aggressively, because checkpoint-heavy training can eat storage far faster than intuition suggests.**
    - BLOOM-176B's checkpointing consumed ~18.4TB of disk *per day*, and on a shared cluster with other jobs also writing data, that burn rate can blow through a "should be plenty" allocation fast.
    - The `fs-watchdog.py` tool monitored both raw disk usage percentage *and* inode usage across every relevant partition, since these are two independent ways to run out of "space."
    - Alert thresholds need to be tuned per-partition, not set to a single blanket percentage - what matters is translating the percentage into "how many days of runway at my current daily burn rate."
    - This is what turns "we ran out of disk space at 3am and lost the run" into "we got an email three days ago and were able to add more storage in time."

📖 Low disk space alerts

- 6.6. **80% can be the new 100% on some distributed filesystems, so `df` showing free space is not a reliable safety guarantee.**
    - Distributed storage is built from many physical disks (each typically only 0.3-2TB); if rebalancing isn't performed often enough, any single disk can hit 100% full and reject writes even while aggregate `df` reports a comfortable 80% used.
    - This precise failure mode was hit during IDEFICS-80B training, where 80% genuinely became the practical ceiling on that filesystem.
    - It varies by technology: Lustre is known to have this issue (80% reliable), GPFS providers claim they don't have it.
    - **Safe move:** ask your storage provider what percentage of nominal capacity they consider reliably usable, and provision with that real ceiling in mind.

📖 Low disk space alerts

- 6.7. **Sometimes the right engineering decision is to work around a leak instead of chasing it to its root cause, especially under time pressure.**
    - During IDEFICS-80B, a slow CPU memory leak took multiple days to build up to an OOM crash - a timescale that makes root-causing painfully slow to iterate on.
    - Instead of halting training to hunt for the source, the team built an in-loop watchdog that monitored resident memory and voluntarily exited once a threshold was crossed, letting the next queued job resume with memory reclaimed.
    - **Concrete pattern:** check `psutil.virtual_memory().percent` (or the GPU equivalent via `torch.cuda.mem_get_info()` after `gc.collect()` + `torch.cuda.empty_cache()`) once per iteration, and exit early past a threshold - cheap insurance worth adding regardless of whether you currently suspect a leak.

📖 Dealing with slow memory leaks

- 6.8. **Handle forced preemption gracefully: know exactly how much runway your program needs to save-and-exit, and tell it that number up front.**
    - SLURM jobs on shared HPC clusters have hard time limits (a common one is ~20 hours, though BLOOM-176B's jobs ran up to 100 hours), and just letting the job get killed wastes

steps since the last checkpoint.

- **BLOOM-176B's Approach A** (via `--exit-duration-in-mins`): tell the program at launch exactly how many minutes it has. It starts an internal timer and checks before each iteration (via an `all_reduce`-synchronized flag across all ranks) whether the budget is exceeded - if so, it saves and exits cleanly before SLURM's hard kill.
- **Sizing the budget:** know how long one iteration takes and how long a checkpoint save takes, sum those, then double for safety margin - exiting early costs nothing, exiting late costs the whole checkpoint.
- **Approach B:** a signal-based approach (`--signal=B:USR1@<seconds>`, trapped directly or relayed through the launcher for `torchrun`/`accelerate`/`deepspeed`), useful when a fixed time budget can't be cleanly computed in advance.

📖 Approach A: tell the program at launch time when to start exiting

7. Training Stability

- 7.1. **Initialization std matters enormously for stability, and getting it wrong can make a large model simply unable to get past the first few thousand iterations.**
    - In the pre-BLOOM 104B experiments, the team couldn't break past early training until they realized Megatron-LM's default at that time `--init-method-std` of 0.02 was far too large for a model of that hidden size.
    - Two published formulas were considered: "Transformers without Tears" (`sqrt(2/(NHIDDEN*5))`) and the 530B paper (`sqrt(1/(NHIDDEN*3))`) - the team deliberately chose the smaller (530B) value as the more conservative option.
    - For BLOOM-176B's `NHIDDEN=14336`, that formula gives `std=0.00482` - a tiny, easy-to-get-wrong number that was one of the key reasons the 176B run never suffered a full divergence.
    - **Lesson:** initialization std should scale with hidden size via a published formula, not be copy-pasted as a fixed constant from a smaller model's config.

    📖 STD Init
- 7.2. **Learn from a failed training rather than just abandoning it: the pre-BLOOM 104B experiments' repeated divergence directly produced the recipe that made BLOOM-176B succeed.**
    - Those 104B attempts kept diverging early on, and post-mortem pointed to two suspects: fp16 (narrower dynamic range, prone to overflow) and training data with a lot of uncleaned web-scraped garbage.
    - For BLOOM-176B, the team changed three things at once: switched fp16 → bf16, used much cleaner data, and added an embedding layer-norm. The combination "made all the difference," yielding an almost perfectly smooth loss curve (one spike, recovered in 200 steps).
    - The point isn't just "use bf16" - a genuinely failed multi-months experiment, properly analyzed rather than written off, directly produced the fix for the next, larger attempt.

    📖 A very failed training

8. Cloud Providers and Procurement

- 8.1. **A large support headcount on a call is not the same thing as real support, and this distinction is worth negotiating into a contract explicitly.**
    - The "we will do our best" pattern often means ten-plus people joining an escalation call with no actual ability to fix the problem - it can take weeks to months (faster with personal

connections) to reach the two people who matter: one product manager, one responsible engineer.
    - Since the customer is legally obligated to keep paying and generally can't unilaterally break the contract, "best effort" is asymmetric unless the contract has specific, quantifiable delivery metrics per critical component.
    - **The most useful remedy to negotiate:** a pause on payment obligations until a defined problem is fixed - the one lever that actually creates an incentive to prioritize the fix.
    - Negotiate this *before* signing - after the fact you have essentially no leverage.

  📖 We-will-do-our-best clause

- 8.2. **Accelerators are what fail, overwhelmingly - roughly 95% of hardware failures on a cluster are the accelerators themselves, not CPUs, storage, or network gear.**
    - This statistic should directly shape fault-tolerance planning: it's not worth equal engineering effort on storage/network resilience as on GPU-failure resilience.
    - **Practical response:** a fast, ideally fully automated replacement path (an API, not a support ticket), plus explicitly asking how many spares the provider keeps and how quickly they commit to swapping a failed unit.
    - Because instant replacement is rarely actually available, the standard mitigation is to pay for ~10% more nodes than needed, potentially using the extras for day-to-day dev work and as instant fallbacks when a production node fails.
    - This overprovisioning is real money, but cheap relative to a stalled multi-hundred-GPU job waiting on a support ticket.

  📖 Dealing with accelerator failures

- 8.3. **The network can quietly steal accelerator memory that you were sold and are paying for, without anyone telling you in advance.**
    - On one cloud, ~5GiB of *each* GPU's memory was already consumed by the provider's networking software before any user workload started - discovered only after signing, not disclosed during sales.
    - As accelerator memory grows (H200's 141GiB, B200's 180GiB usable), a fixed 5GiB overhead matters proportionally less, but on an 80GiB H100 it's a material fraction that can be the difference between a batch size fitting or OOMing.
    - **Actionable:** ask, before signing, exactly how much GPU memory is reserved by the provider's software stack versus how much is actually available, and get the number in writing.

  📖 Does the network steal from the accelerator memory?

- 8.4. **Provider software stacks can trap you on old, unsupported OS versions, purely because their own internal tooling hasn't caught up.**
    - More than once, a provider's required drivers/orchestration/monitoring stack simply didn't support any Ubuntu version newer than some fairly old release, forcing an outdated OS underneath a modern ML stack.
    - Easy to overlook during evaluation (nobody demos "which Ubuntu version can I run") but becomes real friction later: security patches, kernel features, drivers, and some ML framework dependencies can quietly assume a newer base OS.
    - **Mitigation:** ask, during evaluation and not after signing, exactly which OS/software versions are supported and required, and get a commitment on how quickly newer versions will be supported.

  📖 Up-to-date software/OS versions

9. Debugging Methodology

From The Art of Debugging:

- 9.1. **Isolate a failing import or component into a one-liner rather than debugging it inside a large, slow-starting program.**
    - If a large program fails during startup (a failing `import torch`, or a model/tokenizer/config load failing on a missing HF auth token), extract exactly that operation into a standalone one-line command and reproduce it there directly.
    - This matters because fast debug cycles and small reproduction payloads are the two most important prerequisites for successful debugging - waiting ten minutes per attempt across thirty attempts means losing track of which hypothesis you actually tested.
    - One-liners, such as `python -c "import torch"`, are also atomic (searchable in shell history), trivially copy-pasteable to a remote machine, and easy to save as a file of reusable variations.
    - The same technique doubles as how you'd pre-download/cache a large model or dataset in parallel with other development work. `python -c 'import sys; from transformers import AutoTokenizer; AutoTokenizer.from_pretrained(sys.argv[1])' t5-small`

  📖 The power of one-liner programs

- 9.2. **Bisect to find the exact breaking commit rather than manually scanning through hundreds of commits by eye.**
    - Common scenario: pinned to a known-good release, upgraded to bleeding-edge `main` for a new feature, and something broke somewhere across hundreds of commits in between.
    - `git bisect` does a classic binary search: give it one known-good and one known-bad commit, test the midpoint, mark `good`/`bad`, and repeat - 1,024 candidate commits collapse to about 10 rounds of testing.
    - Complications have standard handles: `git bisect skip` for untestable commits, and custom terms (`--term-new=fixed --term-old=broken`) when hunting for a fix rather than a break.
    - Once it converges, the first-bad commit's GitHub page almost always links to the PR that introduced it - an immediate, informed place to ask for help.

  📖 Finding a breaking commit by bisecting revisions

- 9.3. **Get a backtrace from a core file to turn a useless, huge traceback into an exact, actionable line of code.**
    - When a compiled extension (CUDA kernel, NCCL, any C/C++-backed library) segfaults, Python can't produce a meaningful exception at all - you're left with nothing, or an unreadable native dump.
    - **Fix:** `gdb -c /path/to/core /path/to/program` followed by `bt`, which walks the call stack at the moment of the crash and shows the exact source line. `bt full` additionally shows local variable values (e.g. a pointer showing `0x0` reveals it was dereferenced uninitialized).
    - For multi-threaded programs (essentially all serious PyTorch/CUDA programs), `thread apply all bt` shows every thread's stack, since the real root cause sometimes lives in a different thread than the one that crashed.
    - This is exactly how real PyTorch crash reports get root-caused, turning a massive native dump into one actionable line.

  📖 Get the backtrace from the core file

- 9.4. **Backtrace a live, hung process instead of waiting for it to crash or timeout on its own.**
    - Sometimes the problem is a deadlock, not a segfault - the process is alive but never making

progress, so there's no crash event to trigger a core dump.
    - `gdb --pid=<pid>` attaches to an already-running process without restarting it, and `thread apply all bt` shows exactly what every thread is doing - precisely how a real PyTorch deadlock was once diagnosed.
    - If you want to preserve state for later analysis (or free resources immediately), `gcore <pid>` or `kill -ABRT <pid>` forces a core dump, though both also terminate the process.
    - Use this live-attach technique for "spinning" or fully-hung processes, versus the core-file technique above for processes that already crashed.
    - If it's hanging inside Python, `py-spy` is a lifesaver!

  📖 Get the backtrace from the still running process + py-spy

- 9.5. **Run flaky or segfaulting tests directly under gdb rather than trying to reproduce the crash blind.**
    - For an intermittently-segfaulting `pytest` test, launch the whole invocation under gdb from the start (`gdb -ex r --args python -m pytest -sv tests/test_failing.py`) rather than trying to catch it separately afterward.
    - When it segfaults, gdb drops you right back at its prompt at the exact moment. `bt` (or `bt full`) works identically to the core-file case, followed by `c` to continue if desired.
    - This sidesteps the "but it didn't crash when I tried to reproduce it separately" frustration entirely.
    - Combined with the two backtrace techniques above, this is considered sufficient to diagnose roughly 95% of real-world native-level crashes in ML codebases.

  📖 Run the program under gdb

- 9.6. **Missing symbols are very often a link-time or library-version problem, not a mysterious runtime bug - and `nm` lets you actually verify what's present.**
    - `undefined symbol: util_b` at runtime, despite the function existing in your codebase, is frequently caused by the wrong version of a shared library being loaded (e.g. via `LD_LIBRARY_PATH` pointing at an older build).
    - `nm ./your_program | grep symbol_name` shows whether a symbol is defined (`T`) or merely referenced-but-undefined (`U`) in a specific binary or library, turning a guess into a checkable fact.
    - Combined with `ldd` (which shows exactly which library files a binary actually resolved to), you can directly compare `nm` output across two "same-named" library builds and see that one has the symbol and the other doesn't.
    - This combination - `nm` for "does the symbol exist here" and `ldd` for "which file actually got loaded" - resolves the large majority of "missing symbol" runtime errors.

  📖 nm

All lessons are drawn from Machine Learning Engineering and The Art of Debugging by Stas Bekman, distributed under CC BY-SA 4.0.

Book Building

Important: this is still a WIP - it mostly works, but stylesheets need some work to make the pdf really nice. Should be complete in a few weeks.

This document assumes you're working from the root of the repo.

Installation requirements

1. Install python packages used during book build

```
pip install -r build/requirements.txt
```

2. Download the free version of Prince XML. It's used to build the pdf version of this book.

Build html

```
make html
```

Build pdf

```
make pdf
```

It will first build the html target and then will use it to build the pdf version.

Build epub

```
make epub
```

It will first build the html target and then will use it to build the epub version.

Check links and anchors

To validate that all local links and anchored links are valid run:

```
make check-links-local
```

To additionally also check external links

```
make check-links-all
```

use the latter sparingly to avoid being banned for hammering servers.

Move md files/dirs and adjust relative links

e.g. `slurm` => `orchestration/slurm`

```
src=slurm
dst=orchestration/slurm

mkdir -p orchestration
git mv $src $dst
perl -pi -e "s|$src|$dst|" chapters-md.txt
python build/mdbook/mv-links.py $src $dst
git checkout $dst
make check-links-local
```

Resize images

When included images are too large, make them smaller a bit:

```
mogrify -format png -resize 1024x1024\> *png
```

PDF Conversion was done with https://www.princexml.com