
CuTeGen: An LLM-Based Agentic Framework for Generation and Optimization of High-Performance GPU Kernels using CuTe

Tara Saba

Department of Computer Science
University of Toronto
tara.saba@mail.utoronto.ca

Anne Ouyang

Standard Kernel
anne@standardkernel.com

Xujie Si

Department of Computer Science
University of Toronto
six@cs.toronto.edu

Fan Long

Department of Computer Science
University of Toronto
fanl@cs.toronto.edu

Abstract

High-performance GPU kernels are critical to modern machine learning systems, yet developing efficient implementations remains a challenging, expert-driven process due to the tight coupling between algorithmic structure, memory hierarchy usage, and hardware-specific optimizations. Recent work has explored using large language models (LLMs) to generate GPU kernels automatically, but generated implementations often struggle to maintain correctness and achieve competitive performance across iterative refinements. We present **CuTeGen**, an agentic framework for automated generation and optimization of GPU kernels that treats kernel development as a structured generate–test–refine workflow. Unlike approaches that rely on one-shot generation or large-scale search over candidate implementations, CuTeGen focuses on progressive refinement of a single evolving kernel through execution-based validation, structured debugging, and staged optimization. A key design choice is to generate kernels using the CuTe abstraction layer, which exposes performance-critical structures such as tiling and data movement while providing a more stable representation for iterative modification. To guide performance improvement, CuTeGen incorporates workload-aware optimization prompts and delayed integration of profiling feedback. Experimental results on matrix multiplication and activation workloads demonstrate that the framework produces functionally correct kernels and achieves competitive performance relative to optimized library implementations.

1 Introduction

The continued scaling of deep learning models—especially large language models (LLMs)—has made high-performance GPU computing a central bottleneck for modern AI systems. Across workloads in language, vision, and scientific computing, end-to-end throughput is often dominated by a small set of compute-intensive primitives, most notably matrix multiplication and attention/activation operators. While GPU hardware capabilities have advanced rapidly, realizing these gains in practice depends critically on low-level kernel implementations that effectively exploit massive parallelism, the memory hierarchy, and specialized instructions. As a result, system performance is frequently determined not by algorithms alone, but by how close kernel code can approach the hardware’s theoretical limits.

LLM-driven coding agents have reshaped many aspects of software development, yet their impact on high-performance GPU kernels remains limited. For many performance-critical operators, e.g., GEMM and FlashAttention [5], state-of-the-art implementations are still largely hand-engineered by experts. Recent work has explored using LLMs to generate GPU kernels automatically, but the performance gap between generated kernels and carefully tuned expert implementations is still substantial, especially for kernels that rely on hardware-specific features and tightly coupled optimization decisions [7, 11, 16].

A core reason is that efficient GPU kernels are extremely sensitive to low-level design choices. High performance typically requires jointly selecting (i) hardware instructions and tensor-core usage (e.g., MMA/WMMMA variants), (ii) work decomposition and tiling across thread blocks and warps, (iii) explicit management of data movement and caching (e.g., shared-memory layouts and bank-conflict avoidance), and (iv) software pipelining to overlap memory transfers with computation (e.g., staged buffering and latency hiding). These choices are highly interdependent: changing a tile shape affects shared-memory layout, which affects instruction selection and scheduling constraints, which in turn affects occupancy and achievable throughput. This creates a large, coupled search space in which single-shot code generation is unlikely to find a good combination and naive iterative edits often break correctness or regress performance.

CuTeGen: This paper presents CuTeGen, an agentic GPU kernel synthesis system that iteratively generates, evaluates, debugs, and refines kernels via a structured execution-feedback loop. Instead of treating kernel synthesis as a one-time generation problem, CuTeGen treats it as an autonomous optimization process: candidate kernels are compiled, tested against reference outputs, and timed; compilation diagnostics, runtime errors, correctness discrepancies, and performance signals are then fed back to guide subsequent iterations. CuTeGen separates debugging from optimization and emphasizes incremental edits (rather than full rewrites), enabling progressive correction and performance improvement without requiring access to expert-written kernels or manual intervention from experienced GPU developers.

A key design choice in CuTeGen is to generate kernels in CuTe [14], a C++/CUDA template abstraction layer that exposes performance-critical structures such as tiling, layout, and data movement, while providing enough scaffolding to make generation and iterative refinement more stable than in raw CUDA. By operating in CuTe, CuTeGen can more naturally leverage tensor-core-friendly tiling strategies and incorporate memory-pipelining patterns that are central to high-performance GEMM kernels, thereby biasing the search space toward more efficient implementations. Moreover, unlike higher-level domain-specific languages such as Triton [20], CuTe preserves the low-level control needed for further optimization, such as inserting inline PTX to access hardware-specific features.

Finally, to guide performance tuning, CuTeGen integrates hardware profiling using NVIDIA Nsight Compute [15]. Importantly, CuTeGen employs a *delayed profiling integration* technique and does not expose profiling metrics from the outset. For structurally complex kernels such as matrix multiplication, we delay profiling-driven feedback until the code has reached a reasonable baseline through higher-level structural optimization. We find that introducing profiling too early often encourages myopic parameter tuning, such as adjusting tile sizes, before the kernel’s overall structure is sound, thereby increasing the risk of premature convergence to poor local optima. Once the kernel has stabilized, curated profiling summaries are introduced to support targeted refinement.

Results: We evaluate CuTeGen on 12 matrix multiplication kernels and 14 activation kernels from KernelBench [16]. Our results show that CuTeGen achieves an average speedup of $1.70\times$ over the PyTorch reference implementations on the activation kernels. For matrix multiplication, CuTeGen even produces kernels that outperform the reference implementation, which invokes cuBLAS [12], on two benchmark cases.

Contributions: This paper makes the following contributions:

- **CuTeGen:** We present CuTeGen, a novel iterative GPU kernel synthesis framework that generates kernels in CuTe. By operating in CuTe, CuTeGen guides LLMs toward a search space that is richer in efficient kernels, while retaining the low-level control needed for further kernel optimization.
- **Delayed Profiling Integration:** We propose a delayed profiling integration technique that allows CuTeGen to incorporate profiling metrics for performance tuning without prematurely driving kernel generation toward poor local optima.

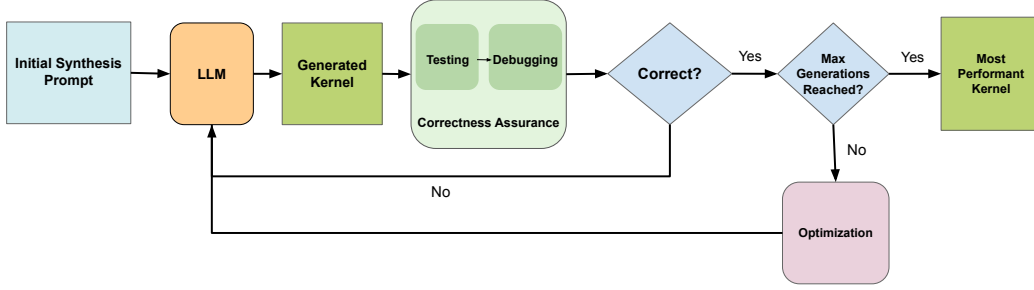


Figure 1: CuTeGen high-level architecture showing iterative kernel generation, correctness assurance, and optimization.

2 Background and Related Work

Kernel Libraries and Compilers. High-performance GPU kernels are typically developed using a stack of vendor libraries, domain-specific frameworks, and compiler systems. Vendor-provided libraries such as cuBLAS [12] and cuDNN [4] offer highly optimized implementations of common operations but are closed-source and limited to predefined kernels. Template-based libraries such as CUTLASS [13], together with its underlying CuTe tensor abstraction layer, expose building blocks for constructing custom kernels while still requiring substantial manual engineering. More recent domain-specific programming systems, including Triton [20] and ThunderKittens [18], simplify GPU programming by providing higher-level abstractions for memory layouts and execution structure, yet kernel implementations must still be written and tuned by developers with less flexibility. Compiler-based systems such as `torch.compile` [17] automate certain transformations at the graph or operator level but generally operate within fixed optimization patterns. These approaches reduce the difficulty of kernel development but still depend on human-designed implementations and optimization strategies.

Benchmarks and Surveys. KernelBench [16] introduces a benchmark suite designed to evaluate whether LLMs can generate correct and high-performance GPU kernels from PyTorch reference implementations. Work presented in [24] provides a systematic overview of the emerging space of LLM-driven kernel generation, categorizing approaches across model training, agentic workflows, and evaluation strategies. Robust kbench [7] further emphasizes the need for reliable correctness verification and hardware-aware evaluation in agentic optimization pipelines.

Empirical Evaluation of LLM Capabilities. Several works focus on empirically evaluating the ability of LLMs to generate HPC kernels. [6] and [21] assess correctness and programming proficiency across classical numerical kernels and programming models. These studies characterize model capabilities but do not propose structured execution-driven refinement frameworks.

Learning-Based and Retraining Approaches. Another line of work improves kernel generation through specialized model training or structured search. K-Search [1] formulates kernel optimization as an LLM-driven planning process over a search tree guided by a co-evolving world model. Ascend-KernelGen [2] and Autotriton [9] improve generation quality through domain-specific datasets and reinforcement learning. CUDA-L1 [10] similarly applies reinforcement learning to enhance CUDA kernel optimization behavior. BabelTower [23] and CodeRosetta [19] instead focus on translating sequential or high-level programs into parallel CUDA implementations. While these approaches demonstrate promising improvements, they rely on expensive retrains or reinforcement learning pipelines with limited performance gains [25]. In contrast, our work focuses on structured refinement driven by execution feedback without requiring additional model training.

Agentic Kernel Optimization Loops. Recent systems most closely related to our work employ agentic generate-evaluate-refine loops. TritonForge [8] and CUDAForge [25] integrate hardware feedback and profiling signals to guide automated kernel optimization. Astra [22] adopts a multi-agent architecture for optimizing kernels starting from a baseline CUDA implementation. CUDA-LLM [3]

and KernelEvolve [11] iteratively improve kernels using compilation and runtime feedback, often exploring multiple candidate implementations or maintaining evolving populations. In contrast, CuTeGen focuses on structured refinement of a single evolving kernel rather than exploring large populations of candidates or repeatedly regenerating full implementations. Our approach emphasizes localized patch edits that preserve previously discovered optimization structure avoiding discarding promising paths. Additionally, CuTeGen targets kernels expressed using the CuTe abstraction layer and incorporates profiling feedback in a curated, stage-dependent manner to guide optimization without exposing the model to raw profiler outputs in all stages.

3 Method

In this section, we describe the design of CuTeGen, its task formulation, and the iterative workflow used to synthesize, validate, and optimize GPU kernels.

3.1 System Overview and Task Formulation

System Overview. Figure 1 presents the overall architecture of CuTeGen. The system operates as an agentic refinement loop in which kernels generated by a large language model are compiled, executed, validated for correctness, and iteratively improved. The workflow is organized into three main phases shown in the figure: **testing**, **debugging**, and **optimization**.

Starting from an initial task specification and reference implementation, the model produces a CuTe-based kernel. This kernel is compiled and executed against the reference implementation using randomized inputs. Compilation failures, runtime errors, and output mismatches are recorded and used to guide subsequent refinement steps. When correctness issues arise, diagnostic information is incorporated into structured debugging prompts that enable the model to analyze failures and generate targeted code patches. Once a kernel produces correct outputs, the system transitions to the optimization phase, where performance feedback obtained through hardware profiling, together with a domain-specific optimization guide that we provide to the model, is used to guide further improvements. Through this iterative feedback loop, CuTeGen progressively refines both functional correctness and execution efficiency.

Task Formulation and Objective. CuTeGen takes as input a PyTorch specification of the target computation, typically written as a `torch.nn.Module` whose `forward` method defines the intended semantics. The user may additionally provide helper code for constructing representative input tensors, mainly to specify expected tensor shapes, data types, and invocation patterns. Based on this specification, CuTeGen synthesizes a functionally equivalent implementation in which performance-critical PyTorch operators are replaced by custom GPU kernels. The output of CuTeGen is therefore not just a generated kernel, but an optimized implementation that can be invoked under the same high-level interface as the original PyTorch code. From the user’s perspective, adopting CuTeGen only requires providing a correct reference implementation together with representative inputs; CuTeGen then handles kernel generation and iterative optimization toward improved execution performance. This task formulation also aligns with standardized evaluation setups used in benchmarks such as KernelBench [16].

In our framework, we explicitly guide the model to generate kernels using the **CuTe** abstraction layer. CuTe provides structured tensor and layout abstractions that expose performance-critical design choices such as tiling strategies, memory layouts, and execution organization. Table 1 compares CuTe with alternative GPU programming abstractions across key optimization capabilities and levels of hardware control. This comparison highlights the role of CuTe as a structured intermediate representation that preserves low-level expressiveness while providing a stable foundation for iterative optimization. Compared to raw CUDA programming, these abstractions make kernel generation more tractable for language models while still enabling low-level control over GPU execution. At the same time, CuTe remains lower-level than higher-level DSLs such as Triton or ThunderKittens, allowing fine-grained control over kernel behavior while constraining the synthesis space in a way that supports iterative optimization.

Table 1: Comparison of GPU programming abstractions across optimization capabilities and control over hardware execution. Entries indicate whether a given abstraction explicitly exposes a capability to the developer or code generator. This comparison highlights the design trade-off between structured guidance and low-level control that motivates our use of CuTe.

Abstraction	Abstraction Level	Explicit Tiling	Shared Memory	Instruction Access	Tensor/Layout Abstractions	Hardware Control
Python (PyTorch)	Very High	No	No	No	No	Low
Triton	High	Partial	Abstracted	Limited	Yes	Medium
CuTe	Structured Low	Yes	Yes	Yes	Yes	High
CUDA	Low	Yes	Yes	Yes	No	Very High
PTX	Assembly	Yes	Yes	Yes	No	Maximum

3.2 Correctness Assurance Component

CuTeGen follows an iterative generate–test–refine workflow in which kernel correctness is established before performance optimization is attempted. The correctness assurance component coordinates a sequence of LLM interactions and program executions designed to progressively identify and repair errors in generated kernels.

The process begins with an *initial synthesis prompt* (depicted in Figure 2), which describes the target task and provides a representative CuTe kernel example to anchor the expected structure and coding style. Using this prompt, the LLM produces an initial candidate kernel implementation. The generated kernel is then evaluated within the *correctness assurance component* (shown in Figure 3). In the *testing phase*, the system repeatedly compiles, executes, and validates the implementation against reference PyTorch outputs using randomized inputs. Compilation failures produce compiler diagnostics, while successful executions are checked for output equivalence with the reference implementation. Any compilation errors, runtime failures, or correctness discrepancies detected during testing trigger the *debugging process*.

When errors are detected, CuTeGen invokes a structured debugging interaction organized as a two-stage process: (i) diagnosis and (ii) repair. This separation reflects our design goal of encouraging explicit reasoning about failures before modifying the kernel implementation.

Diagnosis Stage. In the diagnosis stage, the model receives diagnostic information—including compilation errors, runtime exceptions, and correctness discrepancies—together with our structured debugging guidelines, and is prompted to generate *diagnostic suggestions* explaining the likely causes of the observed failures. We developed these guidelines based on both domain knowledge of CuTe kernel development and empirical observations of common failure patterns in LLM-generated kernels. Accordingly, they encode both CuTe-specific considerations and recurrent LLM pitfalls, providing targeted guidance for interpreting compiler feedback, runtime behavior, and correctness mismatches. The full debugging guideline used in our system is provided in Appendix A.

Repair Stage. Following diagnosis, the model generates *structured patch edits* that repair the implementation. Rather than regenerating the entire kernel, the model proposes localized modifications in a constrained patch format (e.g., line-level insertions, deletions, or replacements), which are then applied directly to the existing code. This staged design encourages the model to first reason explicitly about the source of failure before proposing edits, leading to more targeted and coherent fixes. Constraining the repair process to localized patches further preserves valid portions of the

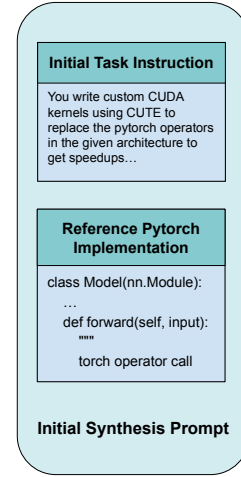


Figure 2: Initial synthesis prompt used to generate the first candidate kernel.

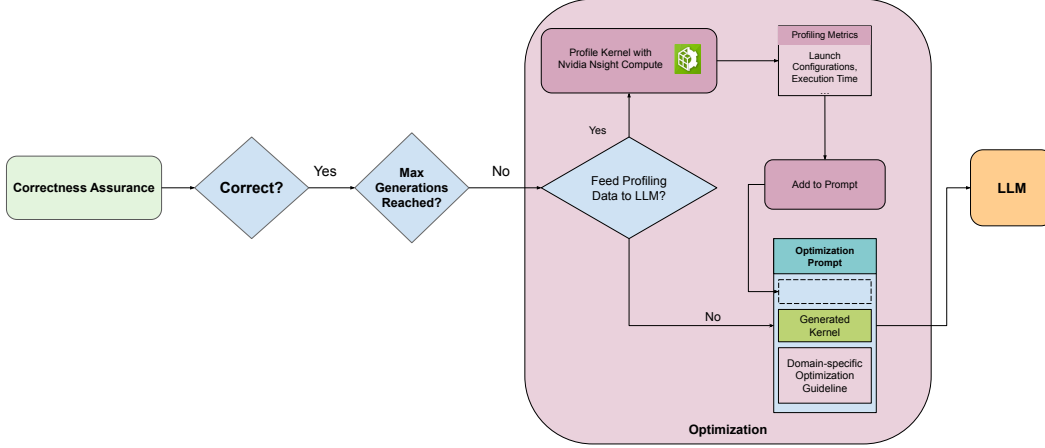


Figure 4: Detailed view of the optimization component. Profiling metrics are incorporated into the prompt when enabled to guide performance-aware transformations. Surrounding stages are omitted for clarity.

Appendix C. In practice, this structured and category-aware guidance enables the model to prioritize meaningful algorithmic improvements over superficial code changes.

Profiling-Driven Refinement. In the profiling-driven refinement phase, CuTeGen incorporates hardware profiling signals obtained using NVIDIA Nsight Compute to inform later optimization stages. Rather than exposing raw profiler outputs, we provide the model with a structured summary of performance indicators, including kernel launch configuration parameters (e.g., block and grid sizes, register usage, and shared-memory allocation), execution-time metrics such as kernel duration and throughput estimates, resource utilization measures including achieved occupancy and memory throughput, and selected profiler diagnostics highlighting potential bottlenecks. Presenting these signals in a concise, structured form enables the model to reason about performance trade-offs without being overwhelmed by low-level detail.

A key design choice in our system is the *delayed introduction of profiling feedback*. We do not expose profiling signals uniformly across all workloads. Instead, for matrix multiplication kernels, profiling information is intentionally withheld during early optimization iterations and introduced only after the model has explored higher-level structural transformations. We found that providing profiling feedback too early biases the model toward premature parameter tuning (e.g., tile sizes or shared-memory configurations), often leading to suboptimal local optima before more impactful algorithmic improvements are implemented. By delaying profiling, we encourage the model to first establish a strong structural design and only then refine low-level parameters.

In contrast, for activation-function kernels—where optimization opportunities are less structurally complex and primarily parameter-driven—profiling feedback is introduced earlier to facilitate efficient tuning. This workload-dependent scheduling of profiling signals allows CuTeGen to balance global algorithmic exploration with targeted performance refinement.

4 Experiments

4.1 Experimental Setup

Benchmarks. We evaluate CuTeGen on a subset of Level-1 workloads from KernelBench benchmark suite, focusing on matrix multiplication and activation function kernels. Matrix multiplication workloads are central to modern AI systems and pose significant optimization challenges due to their sensitivity to memory hierarchy utilization, tiling strategies, data movement patterns, and specialized hardware instructions such as tensor cores. Activation functions, while computationally simpler, are ubiquitous in deep learning pipelines and can be performance-critical due to their frequent invocation.

(a) Matrix multiplication kernels		(b) Activation function kernels	
Kernel	Speedup	Kernel	Speedup
Square MatMul	1.16	ReLU	1.01
Standard MatMul	0.67	Leaky ReLU	1.00
Batched MatMul	0.53	Sigmoid	1.00
Matrix-Vector Multiply	0.86	Tanh	1.00
Matrix-Scalar Multiply	1.02	Softmax	0.88
Large- k MatMul	0.81	LogSoftmax	0.87
Small- k MatMul	0.98	Swish	2.45
Irregular-Shape MatMul	0.82	GELU	1.01
Upper-Triangular MatMul	0.71	SELU	0.99
Lower-Triangular MatMul	1.00	Hard Sigmoid	0.94
3D Tensor MatMul	0.43	Softplus	1.01
MatMul with Diagonal Matrices	17.66	Softsign	3.45
		ELU	0.99
		HardTanh	0.99

Table 2: Performance of CuTeGen-generated kernels. Speedup is reported relative to the corresponding PyTorch reference implementation from KernelBench (> 1 indicates faster execution).

Together, these workloads provide a representative setting for assessing the effectiveness of automated GPU kernel synthesis across both structurally complex and lightweight operations.

Hardware and Software Environment. All experiments were conducted on a workstation equipped with a single NVIDIA GeForce RTX 4090 GPU (24 GB memory) with CUDA 13.0 support. Generated kernels were compiled using PyTorch version 2.8.0 through its CUDA extension interface together with CUTLASS v4.3.0 (commit acb4593), which provides the CuTe abstraction layer used for kernel synthesis. Kernel generation and refinement were performed using GPT-5 within the CuTeGen agentic workflow. All evaluations were performed under identical hardware and software conditions for both baseline and generated implementations.

Evaluation Protocol. Performance is reported as speedup relative to the task’s reference PyTorch implementation (the Model provided in the benchmark task specification), computed as the ratio of reference runtime to generated kernel runtime. Candidate kernels were first required to compile successfully and pass numerical correctness checks against the reference outputs before performance evaluation. Runtime measurements were obtained by averaging execution time over multiple runs under identical hardware and software conditions for both baseline and generated kernels to ensure a fair comparison.

Benchmarking Considerations. Accurate evaluation of generated GPU kernels requires careful benchmarking setup. In particular, KernelBench workloads must be executed under controlled conditions to avoid measurement artifacts arising from warm-up effects, kernel launch overheads, or inconsistent input configurations. We ensure that all kernels are evaluated on identical input sizes and data distributions as specified by the benchmark, and that sufficient warm-up iterations are performed prior to timing to amortize initialization and caching effects. Additionally, we enforce strict correctness validation before any performance measurement, preventing invalid or partially correct kernels from skewing reported results.

A further important consideration is avoiding unintended optimizations outside of the generated kernels themselves. Modern deep learning frameworks and compiler stacks may introduce implicit optimizations such as operator fusion, kernel caching, or backend-specific acceleration that can obscure the true performance of the synthesized kernel. To ensure a fair comparison, we carefully structure our evaluation to isolate the execution of the generated kernel and prevent external optimizations from influencing measured performance. These precautions are essential for obtaining reliable and reproducible comparisons between generated kernels and optimized PyTorch baselines.

4.2 Experimental Results

Performance Evaluation. Table 2 summarizes the performance of kernels generated by CuTeGen across matrix multiplication and activation-function workloads, respectively, reported as speedup relative to the reference PyTorch implementations. Overall, the results indicate that our agentic synthesis workflow can produce competitive GPU kernels, with particularly strong improvements for several activation functions and structured matrix multiplication variants.

For matrix multiplication workloads, performance relative to PyTorch backends is encouraging given the maturity of existing vendor-optimized implementations. While dense GEMM cases remain challenging, CuTeGen achieves comparable performance in several configurations and substantial speedups for certain structured variants such as diagonal or specialized matrix forms. On average across evaluated matmul tasks, performance remains broadly competitive with the reference implementations, suggesting that automated CuTeGen-based synthesis can approach optimized library performance without manual kernel engineering.

Activation-function kernels show more consistent gains, with several cases exhibiting significant speedups. These workloads typically involve simpler computational structure and more predictable memory access patterns, which appear well suited to the incremental optimization workflow employed by CuTeGen. Overall, these results suggest that agentic kernel synthesis may be particularly effective for moderately complex GPU workloads while still yielding competitive performance on more demanding operations.

We do not observe fundamental limitations in the framework that would restrict it to the workloads evaluated in this study. The underlying refinement workflow, representation choice, and optimization strategy are not specific to matrix multiplication or activation functions, but apply to a broad class of GPU operators that follow similar execution patterns. As a result, a natural next step is to extend CuTeGen to additional categories of operations, including kernels such as attention mechanisms, reductions, and fused operators, using the same structured refinement process and workload category-aware settings.

Effect of Delayed Profiling Feedback. To further demonstrate the impact of profiling timing, we compare two optimization schedules for the square matrix multiplication kernel: one in which profiling feedback is provided from the beginning of optimization, and one in which profiling feedback is introduced only after several successful optimization iterations.

We define *depth* as one complete optimization iteration after correctness has been established. Each depth (except for depth 0 in which the generation starts) begins with a correct implementation, applies exactly one optimization attempt, and ends only after the modified kernel has been compiled, debugged if necessary, and re-validated for correctness. Depth therefore measures the number of successful optimization steps applied to a kernel. Figure 5 compares these two schedules. When profiling feedback is exposed from the start of depth 1, performance improvements remain limited and relatively unstable across optimization depths. In contrast, delaying profiling feedback until depth 11 produces a more favorable optimization trajectory, with stronger performance gains in later iterations.

This behavior is consistent with the design rationale described in Section 3.3. In the early stages of optimization, performance is often dominated by higher-level structural choices such as tiling strategy, work decomposition, and data movement. Providing profiling signals too early can bias the model toward premature parameter tuning before the kernel structure has stabilized. Delaying profiling instead allows the model to first establish a correct and structurally stronger implementation, after which profiler feedback becomes more informative for low-level refinement.

These results support our workload-dependent profiling strategy: for structurally complex kernels such as matrix multiplication, delayed profiling leads to more effective optimization behavior, whereas simpler kernels with more limited structural design spaces can benefit from earlier profiling feedback.

5 Case Study: Square Matrix Multiplication

We illustrate CuTeGen’s synthesis behavior through a representative case study: *square* matrix multiplication, where the input matrices are square and of equal dimension (i.e., $A \in \mathbb{R}^{N \times N}$ and $B \in \mathbb{R}^{N \times N}$). Square GEMM is a canonical GPU workload and serves as a useful reference point because

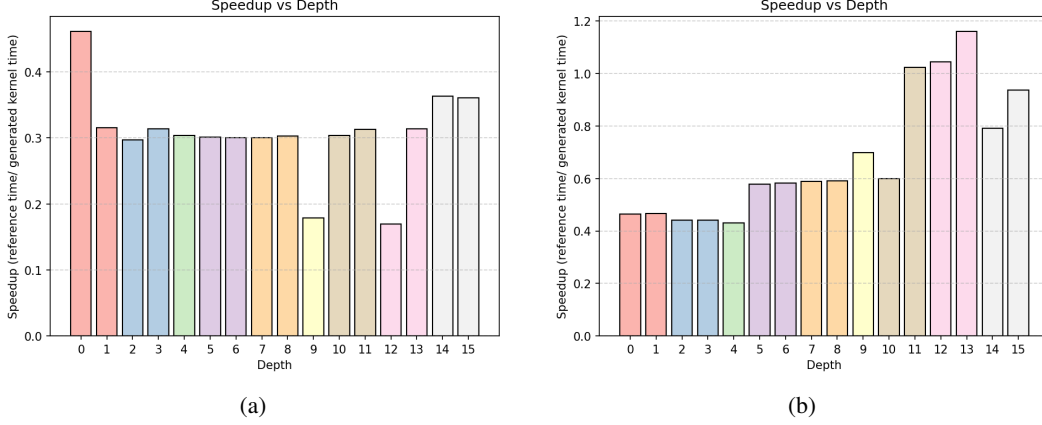


Figure 5: **Effect of profiling timing on optimization behavior.** (a) Profiling feedback provided from the initial optimization iteration (depth 1). (b) Profiling feedback introduced after structural optimization (depth 11). Delaying profiling produces a stronger optimization trajectory and improved final performance.

it exposes the full complexity of high-performance matrix multiplication—including hierarchical tiling, shared-memory staging, asynchronous memory movement, and tensor-core utilization—while avoiding confounding effects from highly irregular shapes. The example below highlights how CuTeGen produces an implementation that matches established GEMM design patterns and how CuTe’s structured tensor abstractions help guide the model toward correct indexing, consistent tiling, and stable iterative refinement.

Hierarchical tiling and work decomposition. The generated kernel adopts a multi-level tiling strategy. At the threadblock level, each CUDA thread block (256 threads, organized as eight warps) computes a fixed output tile of 128×128 elements and iterates over the reduction dimension in K -tiles of size 16, as shown in Figure 6. Within each block, warps are arranged as a 2×4 grid, where each warp computes a 64×32 output region. Finally, each warp decomposes its region into 16×16 tensor-core micro-tiles, consistent with the WMMA fragment configuration illustrated in Figure 8. The tile sizes, warp decomposition, and tensor layouts are internally consistent across shared-memory staging, WMMA fragment dimensions, and global-memory indexing, ensuring correct dimensional alignment and stable execution. This hierarchy (threadblock tile \rightarrow warp tile \rightarrow tensor-core tile) mirrors standard high-performance GEMM implementations and is generated automatically within the CuTeGen refinement loop.

```
using namespace cute;
// Threadblock (CTA) tile: (Mtile, Ntile, Ktile)
auto cta_tiler = make_shape(Int<128>{}, Int<128>{}, Int<16>{});
auto cta_coord = make_coord(blockIdx.x, blockIdx.y, _);

// Global tensors: A(M,K), B(N,K) as a transposed view, C(M,N)
Tensor mA = make_tensor(make_gmem_ptr(A), make_shape(M, K), dA);
Tensor mB = make_tensor(make_gmem_ptr(B), make_shape(N, K), dBt);
Tensor mC = make_tensor(make_gmem_ptr(C), make_shape(M, N), dC);

// Tiles corresponding to this block
Tensor gA = local_tile(mA, cta_tiler, cta_coord, Step<_1, X, _1>{});
Tensor gB = local_tile(mB, cta_tiler, cta_coord, Step<X, _1, _1>{});
Tensor gC = local_tile(mC, cta_tiler, cta_coord, Step<_1, _1, X>{});
```

Figure 6: **Threadblock tiling and CuTe tensor views (square GEMM).** The generated kernel assigns each block a 128×128 output tile and iterates over K in tiles of 16. CuTe shape/stride tensor views make dimensional intent explicit and help stabilize iterative refinement.

Double-buffered shared-memory staging. To reduce global-memory latency and improve overlap between memory movement and compute, the generated kernel allocates two shared-memory stages for both A and B tiles, as illustrated in Figure 7. While tensor-core computation proceeds on one stage, the next K -tile is prefetched into the alternate stage. The shared-memory layouts incorporate padding (“skew”) along leading dimensions to mitigate bank conflicts and preserve 16-byte alignment for vectorized memory transactions. Through iterative debugging and optimization, the model converged on a configuration in which shared-memory strides, tile dimensions, and WMMA fragment sizes remain mutually consistent (Figure 8), avoiding the indexing and layout mismatches commonly observed in naïve CUDA generation. The resulting structure reflects non-trivial performance engineering decisions typically associated with hand-tuned kernels.

```
constexpr int K_TILE = 16;
constexpr int SKEW_A = 8, SKEW_B = 8;
constexpr int SMEM_A_STRIDE = K_TILE + SKEW_A; // padded leading dimension
constexpr int N_TILE = 128;
constexpr int SMEM_B_STRIDE = N_TILE + SKEW_B; // padded leading dimension

extern __shared__ __align__(16) unsigned char smem[];

// Double-buffered shared memory: A0,B0,A1,B1
float* smA0 = reinterpret_cast<float*>(smem);
float* smB0 = reinterpret_cast<float*>(smA0 + 128 * SMEM_A_STRIDE);
float* smA1 = reinterpret_cast<float*>(smB0 + K_TILE * SMEM_B_STRIDE);
float* smB1 = reinterpret_cast<float*>(smA1 + 128 * SMEM_A_STRIDE);

// CuTe shared-memory tensor views with explicit strides (skew/padding)
Tensor sA0 = make_tensor(make_smem_ptr(smA0),
                          make_shape(Int<128>{}, Int<K_TILE>{}),
                          make_stride(Int<SMEM_A_STRIDE>{}, Int<1>{}));
Tensor sB0 = make_tensor(make_smem_ptr(smB0),
                          make_shape(Int<K_TILE>{}, Int<128>{}),
                          make_stride(Int<SMEM_B_STRIDE>{}, Int<1>{}));
```

Figure 7: **Double-buffered shared-memory staging with skewed strides.** Two stages are allocated for both A and B tiles to enable pipelined prefetching. Padding (skew) reduces bank conflicts and helps preserve 16-byte alignment for vectorized transactions (e.g., `cp.async` of 16 bytes).

Inline PTX and asynchronous global-to-shared copies. A particularly notable aspect of the generated implementation is its correct emission of architecture-specific inline PTX instructions to implement asynchronous global-to-shared memory transfers. As shown in Figure 9, the kernel defines device-side wrappers around the SM80+ `cp.async` instruction, issuing 16-byte transactions together with explicit commit and wait-group synchronization. These instructions integrate directly with the double-buffered staging mechanism described above, enabling pipelined data movement across K -tiles. Importantly, the implementation includes architecture guards via `__CUDA_ARCH__` checks, falling back to a synchronous vectorized copy (e.g., `float4`) on pre-SM80 GPUs. This demonstrates that the agent is capable of synthesizing low-level assembly-backed primitives correctly while preserving portability across hardware generations.

```
namespace wmma = nvcuda::wmma;
constexpr int WMMA_M = 16, WMMA_N = 16, WMMA_K = 8;

// Accumulators cover a 64x32 warp tile (4x2 WMMA tiles)
wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> c_frag[4][2];
for (int i = 0; i < 4; ++i)
    for (int j = 0; j < 2; ++j)
        wmma::fill_fragment(c_frag[i][j], 0.0f);

// Compute on the current K tile in steps of WMMA_K
for (int kk = 0; kk < K_TILE; kk += WMMA_K) {
    for (int i = 0; i < 4; ++i) {
        int a_row = warp_m_base + i * WMMA_M;
```

```

wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K,
                wmma::precision::tf32, wmma::row_major> a_frag;
wmma::load_matrix_sync(a_frag, baseA_smem + a_row * SMEM_A_STRIDE + kk,
                      SMEM_A_STRIDE);

for (int j = 0; j < 2; ++j) {
    int b_col = warp_n_base + j * WMMA_N;
    wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K,
                wmma::precision::tf32, wmma::row_major> b_frag;
    wmma::load_matrix_sync(b_frag, baseB_smem + kk * SMEM_B_STRIDE + b_col,
                      SMEM_B_STRIDE);

    wmma::mma_sync(c_frag[i][j], a_frag, b_frag, c_frag[i][j]);
}
}
}

```

Figure 8: **Tensor-core compute loop using WMMA (TF32).** Each warp computes a 64×32 region using $16 \times 16 \times 8$ WMMA operations. The loop iterates over the K -tile in two steps of 8 and accumulates results in registers before storing to global memory.

Early CuTe-only kernel generation. Before converging to the optimized implementation described above, the initial kernel generated by CuTeGen relied exclusively on CuTe abstractions without explicit tensor-core WMMA primitives, inline PTX instructions, or asynchronous memory staging. An excerpt from this early version is shown in Figure 10. The kernel expresses the GEMM computation entirely through CuTe tensor views, tiling constructs, and high-level operations such as copy, gemm, and axpby.

Although this version achieved only moderate performance (approximately $0.45\times$ the reference PyTorch implementation), it was functionally correct and exhibited consistent dimensional reasoning across global-memory layouts, shared-memory tiling, and thread-level partitioning. This CuTe-based formulation provided a stable baseline from which subsequent optimization iterations introduced tensor-core WMMA instructions, asynchronous memory transfers, and double-buffered shared-memory staging, ultimately producing the optimized kernel discussed earlier with a speedup of approximately $1.16\times$.

```

#if __CUDA_ARCH__ >= 800
__device__ __forceinline__ void cp_async_16(void* smem_ptr, const void* gmem_ptr) {
    unsigned smem_addr = static_cast<unsigned>(__cvta_generic_to_shared(smem_ptr));
    asm volatile("cp.async.cg.shared.global [%0], [%1], 16;" :: "r"(smem_addr), "l"(
        gmem_ptr));
}
__device__ __forceinline__ void cp_async_commit() {
    asm volatile("cp.async.commit_group;");
}
__device__ __forceinline__ void cp_async_wait_all() {
    asm volatile("cp.async.wait_group 0;");
}
#else
__device__ __forceinline__ void cp_async_16(void* smem_ptr, const void* gmem_ptr) {
    *reinterpret_cast<float4*>(smem_ptr) = *reinterpret_cast<const float4*>(gmem_ptr);
}
__device__ __forceinline__ void cp_async_commit() {}
__device__ __forceinline__ void cp_async_wait_all() {}
#endif

```

Figure 9: **Architecture-gated asynchronous copy helpers.** For SM80+, the kernel uses inline PTX `cp.async` to copy 16 bytes per instruction from global to shared memory and synchronizes via `commit/wait` group operations. For pre-SM80 targets, it falls back to a synchronous 16-byte vectorized copy and treats `commit/wait` as no-ops.

Role of CuTe abstractions in guiding correct generation. More broadly, CuTe functions as a structured intermediate representation that constrains the synthesis space while still allowing progres-

sively lower-level optimizations. Although the final implementation incorporates WMMA tensor-core primitives and inline PTX asynchronous copy instructions, its overall organization continues to follow the CuTe tensor-view and tiling abstractions illustrated in Figure 6. These abstractions make dimensional relationships explicit—including consistent interpretation of M , N , and K and block-to-tile mappings—reducing ambiguity in indexing and memory layout transformations while supporting incremental optimization without requiring a complete restructuring of the kernel.

```
// Shared memory tiles
__shared__ TA smemA[128*8];
__shared__ TB smemB[128*8];
Tensor sA = make_tensor(make_smem_ptr(smemA), make_shape(Int<128>{}, Int<8>{}));
Tensor sB = make_tensor(make_smem_ptr(smemB), make_shape(Int<128>{}, Int<8>{}));

// Thread layouts
auto tA = make_layout(make_shape(Int<32>{}, Int<8>{}));
auto tB = make_layout(make_shape(Int<32>{}, Int<8>{}));
auto tC = make_layout(make_shape(Int<16>{}, Int<16>{}));

// Partition GMEM tiles for each thread to load to SMEM
Tensor tAgA = local_partition(gA, tA, threadIdx.x);
Tensor tAsA = local_partition(sA, tA, threadIdx.x);
Tensor tBgB = local_partition(gB, tB, threadIdx.x);
Tensor tBsB = local_partition(sB, tB, threadIdx.x);

// Partition SMEM tiles for compute and GMEM C tile for store
Tensor tCsA = local_partition(sA, tC, threadIdx.x, Step<_1, X>{});
Tensor tCsB = local_partition(sB, tC, threadIdx.x, Step<X, _1>{});
Tensor tCgC = local_partition(gC, tC, threadIdx.x, Step<_1, _1>{});

// Accumulator
Tensor tCrC = make_tensor_like(tCgC);
clear(tCrC);

// K loop over tiles
int const K_TILE_MAX = size<2>(tAgA);
for (int k_tile = 0; k_tile < K_TILE_MAX; ++k_tile) {
    // Load A and B tiles from GMEM to SMEM
    copy(tAgA(_,_,k_tile), tAsA);
    copy(tBgB(_,_,k_tile), tBsB);
    __syncthreads();

    // Compute GEMM on the tiles
    gemm(tCsA, tCsB, tCrC);
    __syncthreads();
}
axpy(1.0f, tCrC, 0.0f, tCgC);
}
```

Figure 10: **Initial CuTe-only GEMM kernel generated by CuTeGen.** This early version relies entirely on CuTe tensor abstractions without explicit tensor-core WMMA primitives or asynchronous memory instructions. While functionally correct, it achieves lower performance than the optimized kernel but provides a stable basis for subsequent refinement.

6 Conclusion

We presented **CuTeGen**, a system for automated generation and optimization of GPU kernels that frames kernel development as a structured refinement process driven by execution feedback. Rather than relying on one-shot generation or large-scale search, CuTeGen emphasizes staged improvement: kernels are iteratively compiled, validated, debugged, and optimized using targeted guidance informed by runtime behavior and workload structure.

A central design decision in our system is the use of the *CuTe* abstraction layer as the target representation for kernel synthesis. By exposing performance-critical concepts such as tiling, memory

layout, and execution organization while retaining low-level control over hardware behavior, CuTe provides a practical foundation for incremental optimization. This representation enables the model to reason about kernel structure more consistently than in raw CUDA while still supporting advanced optimizations such as tensor-core usage and asynchronous data movement.

Our empirical evaluation demonstrates that this structured refinement approach can produce functionally correct kernels and achieve competitive performance relative to optimized PyTorch baseline implementations. In particular, we observe consistent improvements for activation workloads and competitive performance across multiple matrix multiplication variants, indicating that iterative, constraint-driven optimization can approach expert-level implementations without requiring model retraining or exhaustive search.

Overall, our findings show that combining structured representations with staged, workload-aware refinement enables stable performance improvement while maintaining correctness throughout the optimization process. These results suggest that automated kernel generation can be integrated into practical development workflows, reducing reliance on manual optimization while preserving strong performance characteristics for performance-critical GPU workloads.

References

- [1] Shiyi Cao, Ziming Mao, Joseph E Gonzalez, and Ion Stoica. K-search: Llm kernel generation via co-evolving intrinsic world model. *arXiv preprint arXiv:2602.19128*, 2026.
- [2] Xinzi Cao, Jianyang Zhai, Pengfei Li, Zhiheng Hu, Cen Yan, Bingxu Mu, Guanghuan Fang, Bin She, Jiayu Li, Yihan Su, et al. Ascendkernelgen: A systematic study of llm-based kernel generation for neural processing units. *arXiv preprint arXiv:2601.07160*, 2026.
- [3] Wentao Chen, Jiace Zhu, Qi Fan, Yehan Ma, and An Zou. Cuda-llm: Llms can write efficient cuda kernels. *arXiv preprint arXiv:2506.09092*, 2025.
- [4] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.
- [5] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.
- [6] William Godoy, Pedro Valero-Lara, Keita Teranishi, Prasanna Balaprakash, and Jeffrey Vetter. Evaluation of openai codex for hpc parallel programming models kernel generation. In *Proceedings of the 52nd International Conference on Parallel Processing Workshops*, pages 136–144, 2023.
- [7] Robert Tjarko Lange, Qi Sun, Aaditya Prasad, Maxence Faldor, Yujin Tang, and David Ha. Towards robust agentic cuda kernel benchmarking, verification, and optimization. *arXiv preprint arXiv:2509.14279*, 2025.
- [8] Haonan Li, Keyu Man, Partha Kanuparth, Hanning Chen, Wei Sun, Sreen Tallam, Chenguang Zhu, Kevin Zhu, and Zhiyun Qian. Tritonforge: Profiling-guided framework for automated triton kernel optimization. *arXiv preprint arXiv:2512.09196*, 2025.
- [9] Shangzhan Li, Zefan Wang, Ye He, Yuxuan Li, Qi Shi, Jianling Li, Yonggang Hu, Wanxiang Che, Xu Han, Zhiyuan Liu, et al. Autotriton: Automatic triton programming with reinforcement learning in llms. *arXiv preprint arXiv:2507.05687*, 2025.
- [10] Xiaoya Li, Xiaofei Sun, Albert Wang, Jiwei Li, and Chris Shum. Cuda-ll: Improving cuda optimization via contrastive reinforcement learning. *arXiv preprint arXiv:2507.14111*, 2025.
- [11] Gang Liao, Hongsen Qin, Ying Wang, Alicia Golden, Michael Kuchnik, Yavuz Yetim, Jia Jiunn Ang, Chunli Fu, Yihan He, Samuel Hsia, et al. Kernelevolve: Scaling agentic kernel coding for heterogeneous ai accelerators at meta. *arXiv preprint arXiv:2512.23236*, 2025.
- [12] NVIDIA. cuBLAS Library. Accessed: [March 2026].
- [13] NVIDIA. CUTLASS: Cuda templates for linear algebra subroutines. <https://github.com/NVIDIA/cutlass>, 2017. Accessed: 2026.
- [14] NVIDIA. Cute dsl. https://docs.nvidia.com/cutlass/latest/media/docs/pythonDSL/cute_dsl.html#cute-dsl, 2026. Accessed: 2026.
- [15] NVIDIA. Nvidia nsight compute. <https://developer.nvidia.com/nsight-compute>, 2026. Accessed: 2026.
- [16] Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? *arXiv preprint arXiv:2502.10517*, 2025.
- [17] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

- [18] Benjamin F Spector, Simran Arora, Aaryan Singhal, Daniel Y Fu, and Christopher Ré. Thunderkittens: Simple, fast, and adorable ai kernels. *arXiv preprint arXiv:2410.20399*, 2024.
- [19] Ali Tehrani, Arijit Bhattacharjee, Le Chen, Nesreen K Ahmed, Amir Yazdanbakhsh, and Ali Jannesari. Coderosetta: Pushing the boundaries of unsupervised code translation for parallel programming. *Advances in Neural Information Processing Systems*, 37:100965–100999, 2024.
- [20] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19, 2019.
- [21] Pedro Valero-Lara, Alexis Huante, Mustafa Al Lail, William F Godoy, Keita Teranishi, Prasanna Balaprakash, and Jeffrey S Vetter. Comparing llama-2 and gpt-3 llms for hpc kernels generation. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 20–32. Springer, 2023.
- [22] Anjiang Wei, Tianran Sun, Yogesh Seenichamy, Hang Song, Anne Ouyang, Azalia Mirhoseini, Ke Wang, and Alex Aiken. Astra: A multi-agent system for gpu kernel performance optimization. *arXiv preprint arXiv:2509.07506*, 2025.
- [23] Yuanbo Wen, Qi Guo, Qiang Fu, Xiaqing Li, Jianxing Xu, Yanlin Tang, Yongwei Zhao, Xing Hu, Zidong Du, Ling Li, et al. Babeltower: Learning to auto-parallelized program translation. In *International Conference on Machine Learning*, pages 23685–23700. PMLR, 2022.
- [24] Yang Yu, Peiyu Zang, Chi Hsu Tsai, Haiming Wu, Yixin Shen, Jialing Zhang, Haoyu Wang, Zhiyou Xiao, Jingze Shi, Yuyu Luo, et al. Towards automated kernel generation in the era of llms. *arXiv preprint arXiv:2601.15727*, 2026.
- [25] Zijian Zhang, Rong Wang, Shiyang Li, Yuebo Luo, Mingyi Hong, and Caiwen Ding. Cudaforge: An agent framework with hardware feedback for cuda kernel optimization. *arXiv preprint arXiv:2511.01884*, 2025.

Appendix

A Debugging Guidelines Used in the Correctness Assurance Component

A central element of the Correctness Assurance Component in CuTeGen is the use of a structured debugging guideline that governs how the model analyzes and repairs incorrect kernels. Rather than relying on unconstrained trial-and-error code generation, our framework provides the model with explicit debugging rules derived from common failure patterns in GPU kernel development and from recurring errors observed in LLM-generated implementations. These guidelines act as a form of operational domain knowledge that shapes the model’s reasoning during the debugging process.

The importance of this guideline arises from the nature of high-performance GPU kernels. Small implementation errors—such as mismatched tensor layouts, incorrect partitioning logic, missing synchronization, or inconsistent tile dimensions—can lead to silent correctness failures, runtime crashes, or performance regressions. In iterative synthesis settings, naïve debugging strategies often attempt to resolve such failures by simplifying the implementation (e.g., removing tiling, bypassing asynchronous pipelines, or replacing custom kernels with library calls). While these changes may restore correctness, they typically destroy the optimization structure of the kernel and prevent meaningful performance improvement. The debugging guideline therefore enforces a disciplined repair strategy that preserves the original algorithmic intent while systematically identifying and correcting the underlying source of failure.

Conceptually, the guideline serves three complementary roles in the system. First, it constrains the model’s search space during debugging by explicitly prohibiting destructive simplifications, ensuring that fixes remain consistent with the intended kernel design. Second, it provides a structured checklist of common correctness pitfalls specific to CuTe and GPU programming, enabling the model to reason about errors using domain-relevant concepts such as tensor shapes and strides, memory layouts, synchronization semantics, and hierarchical tiling. Third, it standardizes the debugging procedure itself by encouraging the model to separate diagnosis from modification, promoting targeted patch edits rather than full kernel rewrites.

The guideline encodes several categories of debugging knowledge. These include rules for preserving kernel structure and optimization intent, explanations of core CuTe abstractions such as tensors, layouts, and tiling hierarchies, and checklists for detecting common correctness issues such as indexing mismatches, synchronization errors, and memory-layout inconsistencies. It also documents common pitfalls related to asynchronous memory operations (e.g., `cp.async`), tensor-core usage, and memory allocation behavior. Together, these components provide the model with a consistent framework for reasoning about failures and generating reliable repairs during iterative refinement.

For reproducibility and transparency, we include below the exact debugging guideline used in all correctness-repair interactions in our experiments.

CuTe Coding and Debugging Guide

CuTe CODING & DEBUGGING GUIDE

NON-NEGOTIABLE RULE (READ FIRST)

```
**DO NOT change the intent of the code by making the CUDA/CuTe code simpler or
↳ by using less CUDA/CuTe.**
- Do **not** ‘fix’ bugs by replacing CuTe tiling/partitioning/TiledMMA/pipeline
↳ with a naive CUDA kernel.
- Preferably do **not** remove async copies, TMA/cp.async, swizzles, tensor
↳ layouts, or CuTe abstractions just to pass correctness.
- Do **not** fall back to cuBLAS / CUTLASS device::Gemm unless the original
↳ intent *already was* to call it.
- Your job is to make the *same algorithm and same CuTe structure* correct:
↳ repair layouts, strides, partitions, synchronization, predication, and
↳ integration wiring-**without de-CuTe-ing the kernel**.
- Do **not** change the input sizes (input tensor sizes) given in the original
↳ model.
```

This is a correctness guide, not a “simplify to green tests” guide.

TABLE OF CONTENTS

PART 0: LLM OPERATING PROCEDURE (How to debug without changing intent)
PART 1: CORE CONCEPTS (Layouts, Shapes/Strides, Tensors, Tiles)
PART 2: ‘HELLO GEMM / HELLO KERNEL’ IN CuTe (Minimal structure you must
↳ preserve)
PART 3: CORRECTNESS DEBUGGING CHECKLIST (Silent bugs & fixes)
PART 4: ASYNC COPY & SYNC (cp.async / TMA / barriers / pipeline gotchas)
PART 5: MMA / TiledMMA PITFALLS (Atom selection, partitioning, layouts)
PART 6: MEMORY & OOM PITFALLS (Especially from redundant contiguous conversions)

PART 0: LLM OPERATING PROCEDURE

When you are given incorrect CUDA/CuTe code, follow this strict workflow:

- 1) ****Freeze intent****
 - * Write down (explicitly, in the response) what the code is trying to
↳ compute:
 - inputs/outputs
 - mathematical operation (e.g., GEMM, convolution-like, reduction,
↳ attention block, stencil, etc.)
 - invariants that must remain true (tiling strategy, pipeline stages,
↳ shared-memory staging, etc.)
 - * Confirm what you will NOT change (see Non-Negotiable Rule above).
- 2) ****Localize the bug without de-optimizing****
 - * You may add:
 - debug prints (guarded)
 - assertions
 - extra verification kernels
 - temporary checksums / spot-check loads
 - temporary extra sync for diagnosis (then remove once fixed)
 - * You may NOT replace the CuTe structure with a simpler kernel “just for
↳ debugging.”
- 3) ****Fix in the smallest possible CuTe-native way****
 - * Typical correct fixes:
 - wrong `Layout` (shape/stride mismatch)
 - wrong tile view (`local_tile`, `tile`, `partition_*`) producing permuted
↳ coordinates
 - missing predicate for edge tiles
 - missing barrier/wait for async copy stages
 - wrong smem layout for the chosen MMA atom
 - wrong accumulator type / epilogue cast timing
 - * Avoid “global refactors.” Make one change, re-test.
- 4) ****Prove the fix****
 - * Provide:
 - a short statement of what was wrong
 - what changed
 - why it preserves intent
 - how it was validated (small cases + at least one edge case)

PART 1: CORE CONCEPTS

- 1) ****Layout = (Shape, Stride) and it is the truth****
 - * In CuTe, `Layout` directly defines address mapping. If output is “plausible
↳ but wrong,” assume layout/stride is wrong until proven otherwise.

- 2) ****Tensor = pointer + Layout****
 * A CuTe `Tensor` is a view. “Correct launch” does not imply “correct indexing.”
 ↪ indexing.”
- 3) ****Tiling/partitioning are coordinate transforms****
 * `local_tile`, `partition_*`, etc. do not copy-they remap indices. Debug
 ↪ them like math.

=====

PART 2: “HELLO GEMM / HELLO KERNEL” IN CuTe (STRUCTURE TO PRESERVE)

=====

A correct CuTe kernel typically has these layers; ****do not delete these**
 ↪ layers**:

- A) ****Global tensors (gmem)****
- B) ****CTA tiling (block-level shapes)****
- C) ****Smem staging + Copy (sync or async)****
- D) ****Compute (MMA or per-thread math)****
- E) ****Epilogue / store (with correct predicates)****

For non-matmul kernels, the same structure still applies: you still have gmem
 ↪ views, tiles, smem staging, and compute that must be consistent.

=====

PART 3: CORRECTNESS DEBUGGING CHECKLIST

=====

Fix in this order:

- A) ****Shape/Layout/Stride sanity****
 1. Confirm each tensor’s real stride (from how data is produced).
 2. Confirm each CuTe view preserves the intended coordinates.
 3. Spot-check a few coordinates → physical addresses → expected values.
 4. Before proposing fixes, check whether the kernel assumes contiguous linear
 ↪ indexing (e.g., `X[i]`, `reinterpret_cast<float4*>`) and whether the wrapper
 ↪ forces `.contiguous()` or changes device; if inputs may be non-contiguous,
 ↪ either (A) require contiguous inputs explicitly (`TORCH_CHECK + contiguity`
 ↪ gate + scalar tail) to match the reference behavior used in this harness, or
 ↪ (B) implement a correct strided fallback (sizes/strides + offset computation)
 ↪ while keeping the optimized contiguous fast path.
- B) ****Copy path correctness****
 4. Validate copy-only loop (`gmem+smem+gmem`) before looking at compute.
 5. If async is used, ensure required waits/barriers exist before first use.
- C) ****Compute correctness****
 6. Validate partition shapes match expectations (especially K mapping for MMA).
 7. Verify accumulator dtype and any casting happens late enough.
- D) ****Edges****
 8. Predication for partial tiles (M/N/K remainders) must be correct.
 9. Ensure stores are masked correctly (no out-of-bounds writes).

High-frequency LLM failure modes (check before anything else):

 - 1) Do NOT worry or check for device mismatches. Our setup is ALWAYS single-gpu.
 - 2) Do NOT assume contiguous/stride/alignment unless you guard it (`TORCH_CHECK` or
 ↪ gated fast path + correct fallback).
 - 3) If using `float4/vec` fast path: gate by `alignment + stride==1`, and correctly
 ↪ handle M not divisible by 4.
 - 4) Do NOT add “fake CuTe” no-ops (unused `make_shape/make_stride`). If CuTe is
 ↪ kept, it must be used for indexing (`cute::Tensor / layouts`), otherwise
 ↪ remove it.

5) Do not use cuBLAS, do not change precision, do not add PyTorch ops in
↳ forward() (must be a single extension call).

PART 4: ASYNC COPY & SYNC

****Async correctness rules (do these before performance tuning):****

- If you introduce `cp.async` / pipelined stages, every stage must have:
 - a clear “produce” point (copy issued)
 - a clear “consume” point (copy completed + visible)
 - correct barrier/wait placement
 - Symptom mapping:
 - stale/previous-tile values → missing wait/barrier
 - fails only when stages > 1 → stage index math or barrier placement bug
 - For debugging you may temporarily serialize:
 - stages = 1
 - copy then compute
 - extra sync
- Then restore original pipeline once fixed.

PART 5: MMA / TiledMMA PITFALLS

- Atom selection must match arch + dtype.
 - Smem layout must match what the MMA path expects (or you must transform).
 - Partitioning must align A/B fragments so K is consistent.
 - Debug rule: prove A/B fragments contain the intended values **before** blaming
↳ MMA.
- ### 4) CuTe GEMM static-assert failures: debug partition **semantics**, not just
↳ types
- If compilation fails inside `cute/algorithm/gemm.hpp` with `static_assert`
↳ errors involving checks like `size<1>(A) == size<1>(B)`, `size<0>(B) ==
↳ size<1>(C)`, `size<1>(B) == size<2>(C)`, or similar, treat this as a
↳ ****fragment-shape mismatch caused by incorrect tiling/partition mapping****. In
↳ practice, this usually means one of the `local_tile(...)`,
↳ `local_partition(...)` , `Step<...>`, or thread-fragment layouts is assigning
↳ the wrong logical role to a dimension. For GEMM, explicitly verify that the
↳ fragments presented to `gemm(...)` obey the contract `A = (M_tile, K_tile)`,
↳ `B = (K_tile, N_tile)`, and `C/Acc = (M_tile, N_tile)`. Do ****not**** just
↳ tweak types blindly. Instead, inspect whether (1) CTA tiling mapped B
↳ through the wrong tiler axis, (2) the shared+compute partition for B
↳ accidentally broadcasts or collapses K/N, or (3) the compute micro-tile
↳ layout (`tC`, accumulator fragment, or MMA fragment shape) is inconsistent
↳ with what CuTe infers. When fixing this class of bug, prefer the smallest
↳ structural correction: repair the `Step<...>` mapping, repair the partition
↳ direction, or make the compute micro-tile match the implied GEMM fragment
↳ shape. After the fix, explicitly state the resulting fragment shapes, e.g.
↳ `tCsA = (16,8)`, `tCsB = (8,16)`, `tCrC = (16,16)`, and confirm that they
↳ satisfy the GEMM contract before recompiling.

PART 6: MEMORY & OOM PITFALLS (IMPORTANT FOR LLMs)

1) The “redundant contiguous() causes OOM” failure mode

Sometimes correctness-check harnesses (especially for non-matmul kernels)

- ↳ accidentally trigger ****multiple redundant contiguous conversions**** and/or
- ↳ large temporary buffers-causing a crash like:

> Runtime error when checking correctness: ****CUDA out of memory. Tried to
↳ allocate 6.00 GiB. GPU 0 has a total capacity of 23.49 GiB of which 3.76 GiB
↳ is free. Process 392481 has 12.44 GiB memory in use. Including non-PyTorch
↳ memory, this process has 6.38 GiB memory in use****

```

**Guideline for the LLM: treat OOM during correctness-check as a bug in the
↪ *test/integration path*, not necessarily in the kernel math.**
Common causes:
- Calling `.contiguous()` on large tensors multiple times in a single check.
- Creating "reference" outputs in multiple dtypes (fp32 + fp16) simultaneously.
- Materializing giant intermediate tensors (e.g., expanded/broadcasted views)
↪ instead of using views.
- Copying tensors to GPU repeatedly inside loops (per-test-case allocation
↪ churn).

### 2) Fix OOM WITHOUT changing kernel intent
You may do any of the following (these preserve intent):
- **Cache contiguous buffers**: if you truly need contiguous, do it once and
↪ reuse.
- **Avoid duplicate references**: compute reference in-place when possible; free
↪ temporaries between runs.
- **Use views instead of materialization**: avoid `.repeat`, `.expand` followed
↪ by implicit materialization.
- **Reduce correctness batch size**: validate on smaller sizes first, then
↪ scale.
- **Explicitly delete temporaries + synchronize** in the harness between checks
↪ when needed.

You must NOT "fix OOM" by rewriting the CuTe kernel into a simpler kernel. The
↪ kernel is not the place to "contiguous away" integration problems.

### 3) A quick diagnostic checklist for this exact error
- Does the harness call `.contiguous()` more than once per input?
- Is there an accidental `.clone()` or `.to(device)` repeated inside a loop?
- Are there multiple large outputs kept alive (not freed) across tests?
- Is the reference path expanding tensors (broadcast) into a full dense
↪ materialization?

=====

END
===

```

B Optimization Prompt

To support the optimization stage described in Section 3.3, CuTeGen uses a structured optimization prompt that conditions the model on the current kernel implementation and the kernel type. The prompt enforces *incremental*, semantics-preserving optimization by requiring the model to implement exactly one optimization attempt at a time while preserving the original code structure, including the algorithm, function signatures, return types, and extension-loading style. This constraint enables performance improvements to be introduced progressively so that each modification can be compiled, validated, and refined in subsequent iterations.

The prompt first instructs the model to classify the kernel as (A) matrix-multiplication/GEMM-like, (B) activation or elementwise, or (C) other. This classification guides the selection of appropriate optimization strategies based on workload structure. For example, GEMM-like kernels may benefit from tiling, shared-memory staging, tensor-core usage, or pipelined data movement, whereas activation and elementwise kernels are typically improved through vectorized memory access, grid-stride execution, lightweight fusion of existing operations, or improved occupancy. Explicitly distinguishing these cases prevents the application of incompatible optimizations, such as tensor-core or split- K transformations to elementwise workloads.

The prompt further constrains the model to preserve functional correctness and avoid invalid shortcuts. It prohibits falling back to PyTorch operators or vendor libraries (e.g., cuBLAS or cuDNN), disallows precision changes, and emphasizes correct indexing, alignment, and bounds handling. For GEMM-like kernels, the model is required to reason about operand dimensions, layouts, and strides before

selecting an optimization and to specialize only when such assumptions are explicitly supported by the reference code or task specification. For elementwise kernels, the prompt instead prioritizes coalesced memory access, correct tail handling, and avoidance of shared memory unless clear data reuse exists.

Overall, the optimization prompt operationalizes the workload-aware refinement strategy described in Section 3.3. By limiting each iteration to a single targeted transformation and combining prompt-guided changes with staged profiling feedback, the system maintains stable optimization behavior while progressively improving kernel performance.

CuTe Optimization Prompt

You're given the following CUTE/CUDA source code:

```
```
<NODE_PRV_SRC>
```
```

Your task: Optimize this code using CUTE/CUDA.

IMPORTANT: Implement ONLY ONE optimization attempt. Pick ONE optimization that
 ↪ is not already implemented.

You must output REAL compilable code (not pseudocode) and preserve the original
 ↪ code structure:

- Keep the same function names, arguments, and return types.
- The optimized output architecture is named ModelNew with custom CUTE/CUDA
 ↪ kernel(s).
- Output ONLY the new CUTE/CUDA code + torch glue code. No extra text, no
 ↪ explanations, no testing code.
- Output the entire new code in ONE CODEBLOCK (wrap in ``` and ```).

Before optimizing, infer what type of kernel this is:

- (A) Matrix Multiply / GEMM-like kernel (e.g., $C = A @ B$, $A * C + A * D$, batched GEMM,
 ↪ etc.)
- (B) Activation / Elementwise kernel (e.g., ReLU, tanh, sigmoid, GELU, add+relu,
 ↪ bias+relu, clamp, etc.)
- (C) Other (if neither fits, apply safe general GPU optimizations only)

Your optimization choice MUST match the kernel type.

=====
 If the kernel is (A) Matrix Multiply / GEMM-like:
 Pick ONE of the following effective optimizations (only one):

- 1) Threadblock-level tiling (CTA tiling for M/N/K)
- 2) Warp-level tiling
- 3) Thread-level tiling / vectorized loads
- 4) Tensor Core mma operations (only if data types + layout allow it)
- 5) Shared-memory bank conflict reduction (swizzling / layout transforms)
- 6) Pipelining / multi-stage loads (cp.async if supported)
- 7) Split-K (only if K dimension is very large AND reduction is bottleneck)
- 8) Asynchronous loads/stores

Rules for GEMM:

- Pay close attention to matrix operand dimensions (M, N, K) and layouts/strides
 ↪ and how they compare to each other to deliver specific optimizations.
- Pick the strongest optimization that matches the shapes.
- There might be information about the matrices in comments or the reference
 ↪ code (such as if a matrix is symmetric etc) you can optimize only for those
 ↪ cases to make kernels fast but DO NOT make assumptions yourself without
 ↪ having provable and explicit information in the reference.
- Do NOT change precision.
- Do NOT use cuBLAS/cuDNN or other pre-optimized libraries.
- Do NOT fall back to PyTorch operators for any case.
- It is acceptable to specialize to the exact shapes listed.

=====
 If the kernel is (B) Activation / Elementwise:

```

Matmul optimizations like tensor cores, split-K, warp tiling for M/N/K, and
↪ matrix swizzling are NOT applicable.
Pick ONE of the following elementwise optimizations (only one):
1) Vectorized global memory loads/stores (float4/half2/etc.) when contiguous +
↪ aligned
2) Grid-stride loop to cover arbitrary numel efficiently
3) Reduce kernel launch overhead: fuse simple elementwise ops already present in
↪ the code (ONLY if they already exist in the original kernel computation; do
↪ not invent new semantics)
4) Use fast math intrinsics safely (only for tanh/sigmoid/GELU-like kernels;
↪ keep outputs correct/stable)
5) Improve tail handling + bounds checks to avoid invalid memory accesses
6) Minimize register pressure / unnecessary temporaries
7) Improve occupancy by choosing a better block size (128/256/512) WITHOUT
↪ changing semantics

Rules for elementwise kernels:
- Treat the tensor as a 1D array of length L = x.numel().
- Memory access must be coalesced.
- Avoid using shared memory unless there is clear data reuse (elementwise
↪ usually has none).
- Keep correct behavior for all elements, including tail elements when L is not
↪ divisible by vector width.
- Do NOT use or fall back to PyTorch ops even for special cases.
- Do NOT change precision.
- It is acceptable to specialize to the exact dtype and contiguous layout if the
↪ original code already assumes it.

=====
If the kernel is (C) Other / unclear:
Apply ONE safe general GPU optimization:
- vectorized load/store OR
- better bounds handling OR
- grid-stride loop
Do NOT introduce GEMM-specific assumptions.

=====
Additional hard constraints:
- Keep the given source code structure the same, including function names,
↪ arguments, return types, and extension loading style.
- Generate complete working code that compiles.
- Do not output multiple versions; implement exactly one optimization attempt.
- Do not add any benchmark or test code.
- Do not add extra commentary outside the code block.
- Pay attention to alignment, indexing correctness, and avoiding out-of-bounds
↪ accesses.

Make sure the optimized code implements the same functionality as the previous
↪ version.

```

C Comparison of CuTe and CUDA Representations for Square GEMM

To better understand the role of the target representation in LLM-based kernel generation, we compare two minimal square matrix multiplication kernels produced under similar prompting conditions: one in standard CUDA and one in CuTe. The goal is not to claim that the initial CuTe kernel is already highly optimized, but to show that it places the model in a more favorable region of the design space from the outset.

As shown in Figure 11, the simplest raw CUDA kernel follows a thread-per-output-element pattern: each thread computes a single output entry using direct global-memory indexing and a full inner-product loop over the reduction dimension. While easy to generate, this form encodes little of the structure required for high-performance GEMM. It lacks explicit notions of tiling, warp-level decomposition, shared-memory staging, and separation between data movement and computation. As a result, moving from this baseline to an efficient implementation requires introducing several

tightly coupled transformations—blocking, mapping to thread hierarchies, shared-memory staging, synchronization, and tensor-core usage—effectively restructuring the kernel rather than refining it incrementally.

```
__global__ void matmul_kernel(const float* __restrict__ A,
                             const float* __restrict__ B,
                             float* __restrict__ C,
                             int M, int K, int N) {
    int row = blockIdx.y * blockDim.y + threadIdx.y; // [0, M)
    int col = blockIdx.x * blockDim.x + threadIdx.x; // [0, N)

    if (row < M && col < N) {
        float sum = 0.0f;
        for (int k = 0; k < K; ++k) {
            sum += A[row * K + k] * B[k * N + col];
        }
        C[row * N + col] = sum;
    }
}
```

Figure 11: **Naïve CUDA implementation of square matrix multiplication.** Each thread computes a single output element using direct global-memory accesses and a full reduction over K . While functionally correct and easy to generate, this form lacks explicit structure for tiling, shared-memory reuse, or hierarchical work decomposition, making it a weak starting point for performance optimization.

In contrast, the simplest CuTe kernel (Figure 12) already exposes the core structural axes of GEMM. It explicitly represents the problem shape (M, N, K) , constructs tensor views, defines a CTA tiler, and partitions computation across blocks and threads, with shared-memory buffers and thread layouts present from the outset. Although not yet optimized, this formulation begins from a tiled and partitioned structure, encouraging the model to reason directly about blocking, decomposition, and data movement. Even at this early stage, this structural advantage translates into measurable performance differences: the initial CuTe kernel achieves a $4.5\times$ speedup over the naïve CUDA baseline (approximately $0.45\times$ vs. $0.1\times$ relative to the PyTorch reference), despite neither implementation being fully optimized.

Crucially, CuTe separates logical tensor structure from its physical mapping onto threads and memory. This reduces reliance on manual index arithmetic and allows modifications to tiling, layouts, and thread mappings without rewriting the entire kernel. In contrast, similar changes in raw CUDA typically require substantial reworking of indexing logic and control flow, increasing the likelihood of correctness regressions.

Overall, enforcing CuTe generation is not merely a syntactic choice. For square GEMM, it changes the form of the initial solution: the raw CUDA kernel is correct but structurally minimal, whereas the CuTe kernel already reflects the abstractions used in high-performance implementations. This provides a stronger and more stable starting point for subsequent optimization.

```
template <class TA, class TB, class TC>
__global__ void
gemm_nn_device(cute::Shape<int,int,int> shape_MNK,
               TA const* A, cute::Stride<int,int> dA,
               TB const* B, cute::Stride<int,int> dBt, // Transposed view stride for
                  B: (N,K) with stride (1,N)
               TC * C, cute::Stride<int,int> dC)
{
    using namespace cute;

    // Unpack shape
    int const M = get<0>(shape_MNK);
    int const N = get<1>(shape_MNK);
    int const K = get<2>(shape_MNK);

    // Global memory tensors
    // A is (M,K) row-major
    Tensor mA = make_tensor(make_gmem_ptr(A), make_shape(M, K), dA);
```



```

// B is given as (K,N) row-major, create a transposed view (N,K) with stride (1, N)
)
Tensor mB = make_tensor(make_gmem_ptr(B), make_shape(N, K), dBt);
// C is (M,N) row-major
Tensor mC = make_tensor(make_gmem_ptr(C), make_shape(M, N), dC);

// Tile shapes: (Mtile, Ntile, Ktile)
auto cta_tiler = make_shape(Int<128>{}, Int<128>{}, Int<8>{});
auto cta_coord = make_coord(blockIdx.x, blockIdx.y, _);

// Tile tensors from GMEM
Tensor gA = local_tile(mA, cta_tiler, cta_coord, Step<_1, X, _1>{}); // (128, 8)
    along M and K
Tensor gB = local_tile(mB, cta_tiler, cta_coord, Step<X, _1, _1>{}); // (128, 8)
    along N and K (B is (N,K))
Tensor gC = local_tile(mC, cta_tiler, cta_coord, Step<_1, _1, X>{}); // (128,128)
    along M and N

// Shared memory tiles
__shared__ TA smemA[128*8];
__shared__ TB smemB[128*8];
Tensor sA = make_tensor(make_smem_ptr(smemA), make_shape(Int<128>{}, Int<8>{}));
Tensor sB = make_tensor(make_smem_ptr(smemB), make_shape(Int<128>{}, Int<8>{}));

// Thread layouts
auto tA = make_layout(make_shape(Int<32>{}, Int<8>{}));
auto tB = make_layout(make_shape(Int<32>{}, Int<8>{}));
auto tC = make_layout(make_shape(Int<16>{}, Int<16>{}));

// Partition GMEM tiles for each thread to load to SMEM
Tensor tAgA = local_partition(gA, tA, threadIdx.x);
Tensor tAsA = local_partition(sA, tA, threadIdx.x);
Tensor tBgB = local_partition(gB, tB, threadIdx.x);
Tensor tBsB = local_partition(sB, tB, threadIdx.x);

// Partition SMEM tiles for compute and GMEM C tile for store
Tensor tCsA = local_partition(sA, tC, threadIdx.x, Step<_1, X>{});
Tensor tCsB = local_partition(sB, tC, threadIdx.x, Step<X, _1>{});
Tensor tCgC = local_partition(gC, tC, threadIdx.x, Step<_1, _1>{});

// Accumulator
Tensor tCrC = make_tensor_like(tCgC);
clear(tCrC);

// K loop over tiles
int const K_TILE_MAX = size<2>(tAgA);
for (int k_tile = 0; k_tile < K_TILE_MAX; ++k_tile) {
    // Load A and B tiles from GMEM to SMEM
    copy(tAgA(_,_,k_tile), tAsA);
    copy(tBgB(_,_,k_tile), tBsB);
    __syncthreads();

    // Compute GEMM on the tiles
    gemm(tCsA, tCsB, tCrC);
    __syncthreads();
}

// Write back to GMEM: C = 1.0 * Acc + 0.0 * C
axpy(1.0f, tCrC, 0.0f, tCgC);
}

```

Figure 12: **Initial CuTe implementation of square matrix multiplication.** Unlike the naïve CUDA baseline, this kernel is already expressed in terms of GEMM-relevant abstractions, including explicit problem shapes, CTA tiling, tensor views, shared-memory tiles, and thread-level partitions. Although still simple and not fully optimized, this representation gives the model a stronger starting point by making blocking, data movement, and hierarchical work decomposition explicit from the outset.