

Accelerating Persistent Scatterer Pixel Selection for InSAR Processing

Tahsin Reza¹, Aaron Zimmer, José Manuel Delgado Blasco²,
Parwant Ghuman, Tanuj Kr Aasawat, and Matei Ripeanu

Abstract—Interferometric Synthetic Aperture Radar (InSAR) is a remote sensing technology used for estimating the displacement of an object on the ground or the earth's surface itself. Persistent Scatterer-InSAR (PS-InSAR) is a category of time series algorithms enabling high resolution monitoring. PS-InSAR relies on successful selection of points that appear stable across a set of satellite images taken over time. This paper presents *PtSel*, a new algorithm for selecting these points, a problem known as *Persistent Scatterer Selection*. The key advantage of *PtSel* over the key existing techniques is that it does not require model assumptions, yet preserves solution accuracy. Motivated by the abundance of parallelism the algorithm exposes, we have implemented it for GPUs. Our evaluation using real-world data shows that the GPU implementation not only offers superior performance but also scales linearly with GPU count and workload size. We compare the GPU implementation and a parallel CPU implementation: a consumer grade GPU offers 18x speedup over a 16-core Ivy Bridge Xeon System, while four GPUs offer 65x speedup. The GPU solution consumes 28x less energy than the CPU-only solution. Additionally, we present a comparison with the most widely used PS-interferometry software package StaMPS, in terms of point selection coverage and precision.

Index Terms—GPU acceleration, image processing, time series analysis, remote sensing, InSAR

1 INTRODUCTION

INSAR is a remote sensing technology primarily used for estimating displacement of the earth's surface over large areas with up to millimeter precision [1], [2]. InSAR has applications in areas such as monitoring earth subsidence and uplift due to urban infrastructure development [3], mining [4], oil and gas extraction [5], and permafrost thawing [6]. Many sophisticated SAR satellites exist and produce data that covers the entire globe including Canadian Radarsat 1/2, European ERS-1/2, ENVISAT, Sentinel 1A/1B, and Japanese ALOS.

As it travels along its orbital path, a SAR satellite emits radar signal towards the earth and records the echo. The earth's surface area is divided into a grid, whose *resolution cells* are in the order of square meters [7]. Each resolution cell corresponds to a single pixel in a 2D radar image. Signal components, e.g., phase and amplitude, in the retuning radar echo from a particular resolution cell contain information that can be used for deformation estimation. SAR imagery based earth deformation estimation operates on images

of the same region-of-interest (i.e., a collection of resolution cells) taken over an extended period of time. Displacement can be estimated from the phase difference in different images, of the same resolution cell, taken at separate points in time [2].

SAR satellites produce *single-look complex* (SLC) images: each pixel in the SLC image is a complex number representing the phase and amplitude of the returning radar echo from a particular resolution cell on the ground. An *interferogram* is created from two temporally separated SLC images via the pointwise product of one SLC image with the complex conjugate of the other SLC image. Thus, each pixel in an interferogram is a complex number indicating the phase difference between the two temporally separated SLC images. This phase difference encodes the surface displacement signal as well as many other contaminant signals (e.g., atmosphere distortion, orbital error, elevation model error, and noise), which can largely be removed through careful filtering. Once these signals are removed, we only know the principal value of the remaining displacement signal (i.e., modulo 2π), so in order to reconstruct the true displacement, a *phase unwrapping* algorithm is applied to the 2π -wrapped phase [8]. Phase unwrapping is the process of correcting surface altitude measurement ambiguity that exists in an interferogram [9], [10].

Persistent Scatterer Interferometry (PSI) represents a specific class of InSAR techniques. One of the challenges of InSAR is increased temporal decorrelation of SAR signals due to surface change, particularly in rural, vegetated areas, which can prevent the recovery of the phase measurement. PSI techniques aim to provide reliable measurements in decorrelated regions as well [11]. PSI based deformation

- T. Reza, T.K. Aasawat and M. Ripeanu are with Electrical and Computer Engineering, University of British Columbia, Vancouver, BC V6T 1Z4, Canada. E-mail: {treza, taasawat, matei}@ece.ubc.ca.
- A. Zimmer and P. Ghuman are with 3vGeomatics, Vancouver, BC V5Y, Canada. E-mail: {azimmer, parwant}@3vgeomatics.com.
- J.M.D. Blasco is with Grupo de Investigación Microgeodesia Jaén, Universidad de Jaén, Jaén 23071, Spain. E-mail: jdblascos@ujaen.es.

Manuscript received 1 Mar. 2016; revised 10 Jan. 2017; accepted 9 May 2017.
Date of publication 19 May 2017; date of current version 8 Dec. 2017.

(Corresponding author: Tahsin Reza.)

Recommended for acceptance by A. Dubey.

For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.

Digital Object Identifier no. 10.1109/TPDS.2017.2706291

measurements relies heavily on estimates of the phase *quality* of pixels: pixels that are more coherent (i.e., they are stable in a region-of-interest over time) than others and are trusted during the unwrapping phase. Selection of these stable pixels, formally known as *Persistent Scatterer* pixels, with high accuracy and efficiency is an important yet challenging problem. One of the goals is to perform well in the presence of *distributed scatterer* (details in Section 2.1) pixels that are a superposition of microwave reflections of multiple objects within a resolution cell. Persistent Scatterer pixels are free of such ambiguities and contain information the phase unwrapping process can use [10].

This paper presents *PtSel*, a new algorithm for persistent scatterer (PS) pixel selection (Section 2.3). The algorithm has been conceived based on one of the characteristics observed in SAR imagery. The intuition is that for each PS pixel, a nearby neighbour pixel can be found with similar displacement signal and topographical error signal. The technique we use to infer PS pixels is based on computing the *temporal coherence* of a pixel in a *network of interferograms* [12]. A network of interferograms is made up of all possible interferometric combination of temporally separated, co-registered SLC images that cover the same area of interest. A pixel whose phases are highly coherent over time is likely to be more stable. Instead of computing temporal coherence on individual pixel phase as in [12], *PtSel* does it on the wrapped phase spatial derivative (i.e., it also considers phase difference between neighbouring pixels in the same SLC image). The maximum of all possible temporal coherences between a given pixel and its neighbours is used as an indicator of a PS pixel. The key advantage of *PtSel* is that it eliminates the need for model assumptions (primarily absence of noisy signal components) as in [12].

Typical InSAR data volumes are in the order of hundreds of gigabyte to terabytes. The search-based nature of *PtSel* leads to high computational demands, and ultimately, to slow processing on general purpose CPUs. This limits the ability to use the *PtSel* algorithm in practice and to integrate it to the mission critical InSAR processing chain, despite its accuracy. *PtSel*, however, exposes high data parallelism: pixels can compute their temporal coherence in parallel. This makes *PtSel* a good candidate for executing it on a massively parallel platform like GPUs that offer much higher rate of floating point operations and off-chip memory access bandwidth than general purpose CPUs (even normalized for dollar or energy cost).

Contributions. This paper summarizes our experience designing, optimizing, and evaluating *PtSel* for NVIDIA GPUs using the CUDA framework. Our key contributions include:

- We present *PtSel*, a new algorithm for persistent scatterer pixel selection (Section 2).
- We characterize the compute and memory behaviour of *PtSel* using the roofline model [13]. We analytically establish platform oblivious performance bounds (Section 2). We then present roofline plots for attainable and sustained performance on CPU and GPU and highlight the performance bounds. This analysis justifies our architecture choice for accelerating *PtSel* factors (Section 5).

- We discuss the design choices and the optimization techniques used to ensure that the implementation is in harmony with the underlying GPU processing model (Section 3).
- Additionally, we present a comparison with the widely-used PSI software package StaMPS [14], in terms of PS selection coverage and precision, for two ENVISAT datasets, representing urban and rural environments respectively (Section 6.4).

Our evaluation aims to answer the following questions:

- Under typical workload, with respect to time-to-solution, how much faster is the GPU implementation compared to a parallel CPU implementation?
- Does the GPU implementation scale with workload size and algorithm specific parameters that directly influence processing intensity?
- Despite having higher power rating than a CPU, does a GPU solution offer superior energy efficiency?

Our evaluation (Section 6) using a real-world workload demonstrates the advantage of GPU acceleration. On a pair of Intel Xeon CPUs (Ivy bridge, 32 cores @2.6 GHz), processing this workload with *PtSel* takes 33 hours. On a single NVIDIA GeForce GTX TITAN GPU, the same task takes just under two hours. On four GPUs, it takes about half hour, offering up to 65x speedup. The GPU implementation scales linearly with GPU count and workload size. Power measurements show a 28x improvement in energy consumption by the GPU solution for the same workload.

Our GPU implementation has thus far been successfully integrated to the production processing chain at 3vGeomatics, an InSAR company based in Vancouver.

2 PERSISTENT SCATTERER SELECTION: BACKGROUND AND THE *PTSEL* SOLUTION

In this section, we discuss the importance and relevance of persistent scatterer selection in InSAR. We review the key contributions found in the literature. We then present a new algorithm dubbed *PtSel*. We characterize the compute and memory requirements of *PtSel* following the well-known roofline model [13].

2.1 Importance of Persistent Scatterer Selection

Persistent scatterer-InSAR (PS-InSAR) is a category of time series algorithms within InSAR [7]. PS-InSAR has the advantage of being able to associate the deformation with a specific scatterer rather than a resolution cell of dimensions dictated by the radar system; enabling very high resolution monitoring [7]. The PS-InSAR technique considers only the ‘stable pixels’ (i.e., *persistent scatterer* or *PS pixels*): pixels in a sequence of temporally correlated interferograms whose phase quality is high throughout all the interferograms. The reason is that other pixels may contain ambiguous phase information: radar echo returning for the resolution cell may be composed of a superposition of microwave reflections off the ground. For instance, a cell may contain a telephone pole and a fire hydrant each with different heights and depending on the orbital position of the satellite the reflections from the two objects may interfere to varying extents resulting in unreliable phase throughout the

interferograms. Such pixels are called *distributed scatterer pixels*. PS pixels on the other hand, are free of such ambiguity and contain useful information [2], [7].

Identifying PS pixels in a SAR image stack is the crucial step with in the PS-InSAR workflow. The functionality and efficiency of the rest of the processing majorly depends on successful identification of PS pixels. A persistent scatterer algorithm assigns each pixel in a SAR image stack a rank, called the *temporal coherence*, which indicates how reliable the information the pixel encodes is.

2.2 Past Work on Persistent Scatterer Selection

Several approaches for PS pixel selection have been proposed in the literature [7], [15]. Often the amplitude is used as a proxy for the phase information and pixels with low amplitude dispersion (i.e., low ratio of temporal amplitude variance to mean amplitude) across the network of interferograms are chosen as PS pixels. This approach works well for high amplitude PS pixels, but not for low amplitude ones [16]. Another approach is to compute the temporal coherence of a pixel through the network of interferograms. However, the phase of an interferogram contains many other signals that bias down the temporal coherence; for instance, a strong displacement signal. Previous approaches would first remove the spatially correlated signal as well as baseline (the separation between satellites collecting data) correlated signal before computing temporal coherence by modeling their phases [12]. This technique has high computational overheads and errors can arise due to inaccurate modeling (e.g., assumptions regarding recorded signal being free of various types of noises).

2.3 PtSel—The Proposed Algorithm

Distributed scatterers are prevalent in rural, high-noise and specular environments and cannot be measured individually. A phenomenon observed in SAR imagery, in the majority of the landscapes and environments, is that for each PS candidate pixel, multiple nearby neighbour pixels can be found with similar signal characteristics. For example, numerous proximal urban targets of similar height are ubiquitous, especially in high-resolution datasets. *PtSel* harnesses this observation.

Formally speaking, *PtSel* assumes that, for each persistent scatterer candidate pixel C , a partner pixel N with similar displacement signal and topological error signal, can be found within a predefined neighbourhood (the *search window*, defined by an $r \times r$ square grid, where r is an odd integer, with the pixel C in the center).

Instead of computing temporal coherence on the individual pixel phase as in [12], *PtSel* computes temporal coherence of the pair of pixels C and N , called an *arc*. To this end, *PtSel* computes the wrapped phase derivative by multiplying one pixel phase by the complex conjugate of the other pixel: $arc = C\bar{N}$.

For example, if neighbouring pixels C and N are similar, e.g., located on the balcony of the same building, the temporal coherence of *arc* will be high. If C and N are dissimilar, e.g., park bench versus grass, the temporal coherence of the *arc* will be low due to temporal decorrelation. Typically, the arcs are very short spanning a few decameters or less. All spatial low-pass signals (deformation, atmosphere, orbital

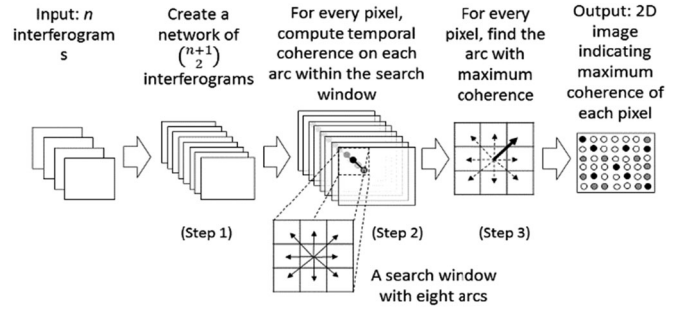


Fig. 1. Illustration of the *PtSel* algorithm depicting the primary steps in order. The n input interferograms are created from $n + 1$ temporally successive SLC images.

error, etc.) cancel over such short distances, i.e., there is negligible coherence reduction due to such signals.

Algorithm. The key steps of *PtSel* are (Fig. 1):

- *Step 1.* *PtSel* starts from a collection of n pre-generated interferograms (created from $n + 1$ SLC images) and forms a network of $\binom{n+1}{2}$ interferograms based on all possible pairwise comparisons.
- *Step 2.* For each pixel (through the interferogram network), *PtSel* computes the temporal coherence on all the arcs within the search window. The temporal coherence (τ) of an arc is computed as:

$$\tau = \left\| \frac{\sum_{k=1}^{\binom{n+1}{2}} C_k \bar{N}_k}{\binom{n+1}{2}} \right\|$$

- *Step 3.* By comparing all arcs, *PtSel* finds the most similar neighbouring pixel that maximizes temporal coherence. The PS indicator is the maximum (τ_{max}) of all possible temporal coherences between a given pixel and its neighbours, since all the factors that bias down the temporal coherence of the candidate PS pixel have largely been cancelled by finding a similar pixel nearby to form an arc with: $\tau_{max} = \max_{0 \leq k < F} (\tau_k)$ where F is the number of neighbours in the search window.

The output of *PtSel* is a 2D image (also called a *coherence map*) where each pixel stores a floating point number in the range (0.0, 1.0] indicating coherence (τ_{max}) of the maximum arc in the input interferograms. The phase unwrapper then uses heuristics to decide if pixels are good PS candidates based on their coherence value.

Generating the Interferogram Network on the Fly. The size of a typical interferogram is in the order of gigabytes. For example, a $15,000 \times 8,000$ pixels interferogram, where each pixel is a complex number (i.e., two single-precision floating point values), is about 1GB of data. A network of interferograms created from $n + 1$ SLC images (or n pre-generated interferograms) will have $\binom{n+1}{2}$ interferograms, thus a network created from 50 interferograms would require about 2.5TB of storage. Pre-generating the interferogram network (often from 100s of interferograms) would require an enormous amount of storage and make in-memory processing infeasible. To address this problem, we combine

interferogram network generation and temporal arc coherence computation. The network is created, on the fly, for each arc within a search window. Thus, a temporal coherence (τ) of an arc is computed using the following equation:

$$\tau = \left\| \frac{\sum_{i=1}^n C_i \bar{N}_i + \sum_{i=1}^n \sum_{j=i+1}^n C_i \bar{N}_i \overline{C_j N_j}}{\binom{n+1}{2}} \right\| \quad (1)$$

Here, i and j are two interferograms. A new arc in the network is computed by taking complex conjugate of two arcs in two interferograms: $C_i \bar{N}_i \overline{C_j N_j}$.

Search Window Size. The size of the search window limits how far *PtSel* looks for similar pixels. For instance, if we want to locate a PS in the middle of an incoherent field, such as a bench in a park, search needs to be performed in a larger neighborhood so that arcs with other benches in the park can be formed (as vegetation movements strongly impact the quality of the radar signal reflected). On the other hand, in urban areas, a smaller search window should be sufficient. To accommodate different datasets, *PtSel* allows user-defined search window size.

2.4 *PtSel* Compute and Memory Requirements Characterization

The computation pattern of *PtSel* is similar to a high-dimensional *stencil* computation on multi-dimensional structured grids [18]. In each time step, a stencil computation updates all the elements in a multidimensional array according to some fixed pattern called a *stencil operator*. In *PtSel*, a pixel together with its neighbours throughout the search window and the input interferograms, forms a stencil. The stencil operator computes maximum temporal arc coherence of a pixel (τ_{max}) using Equation (1).

In this section, we analytically establish platform oblivious performance bounds of *PtSel*. We use the roofline model [13] for the analysis. In the ‘roofline’ vocabulary, the ratio of main memory operations to floating points operations of an application is denoted as the *operational intensity*. We determine operational intensity of *PtSel* for three different cache scenarios, namely, *zero-cache* (OI_0), *infinite-cache* (OI_∞) and *non-zero-cache* (OI_c). Zero- and infinite-cache scenarios represent the two extremes, yielding ‘lower’ and ‘upper’ bound of operational intensity respectively for *PtSel* on any processing platform.

Cache Model. All modern systems incorporate a hierarchical caching mechanism to reduce memory pressure. We are interested in understanding influence of the cache size on the operational intensity. We assume a system having a single level, fully associative cache with capacity c . The cache adopts the least-recently-used (LRU) eviction policy. We assume a fully associative cache and thus the model only considers compulsory and capacity misses (i.e., we assume no conflict misses).

Operational Intensity. We assume a workload that consists of n input interferograms. The width/height of an interferogram is p and a search-window is $r \times r$ pixels. Hence, there are $p^2 \times r^2$ arcs to be computed. Let ζ be the number of computations per arc and ξ be the number of bytes read from the main memory. The operational intensity for *PtSel* can be expressed using the following equation:

TABLE 1
Operational Intensity for Three Cache Scenarios

Zero (OI_0)	Infinite (OI_∞)	Non-zero (OI_c)
1.5	$4 \times n \times r^2$	$OI_0 \times (1/\text{miss-rate})$

$$OI = (p^2 \times r^2 \times \zeta) / \xi$$

An arc formed by two pixels C and N is computed as follows: $arc = C \bar{N} = e^{i(\arg(C) - \arg(N))}$. Here, $\arg(\mathbb{Z}) = \text{atan2}(y, x)$, where $\mathbb{Z}$ is a complex number. $\mathbb{Z} = x + iy = r \cos \phi + ir \sin \phi$ and $r = |\mathbb{Z}| = \sqrt{x^2 + y^2}$ where x and iy are the real and imaginary parts respectively and ϕ is the argument of $\mathbb{Z}$.

Equation (1) computes the temporal coherence (τ) of an arc. In our implementation (without optimization 1 in Section 3.3), `math.h` function `arg` (or `atan2`) is called $n \times 2 + \binom{n}{2} \times 4$ times, `sin` and `cos` each are called $n + \binom{n}{2} \times 2$ times, `sqrt` is called once and an additional $n \times 4 + \binom{n}{2} \times 17$ floating point arithmetic operations are performed. Assuming, all the `math.h` functions do at least four floating point operations per invocation, the total number of floating point operations required to compute temporal coherence of an arc,

$$\zeta = n \times 20 + \binom{n}{2} \times 49 + 4$$

Each pixel in an interferogram is a complex number, where the real and imaginary parts are each 4-byte single-precision floating point number. Computing an arc requires reading in 16 bytes; thus, to solve Equation (1), $n \times 16 + \binom{n}{2} \times 32$ bytes are read. Therefore, total number of bytes read to process the entire workload:

$$p^2 \times r^2 \left(n \times 16 + \binom{n}{2} \times 32 \right)$$

Table 1 summarizes operational intensity for three different cache scenarios. For all three cases, total number of computations (ζ) are the same. However, the number of bytes read from the main memory (ξ) varies according to the size of the cache. In the remainder of this section, we present how we have derived the operational intensity for each of the cache-scenarios.

Zero-cache. Operational intensity for the zero-cache scenario,

$$OI_0 = (p^2 \times r^2 \times \zeta) / \xi$$

When there is no cache,

$$\xi = p^2 \times r^2 \left(n \times 16 + \binom{n}{2} \times 32 \right)$$

i.e., all bytes are read from the main memory. Hence,

$$\begin{aligned} OI_0 &= \left(n \times 20 + \binom{n}{2} \times 49 + 4 \right) / \left(n \times 16 + \binom{n}{2} \times 32 \right) \\ &= (n^2 \times 49 - n \times 9 + 8) / (n^2 \times 32) \\ &\approx (n^2 \times 49) / (n^2 \times 32) = 1.5 \end{aligned}$$

As we can see, OI_0 is independent of n , p and r .

Infinite-cache. For an infinite cache the system will only encounter compulsory misses. Operational intensity,

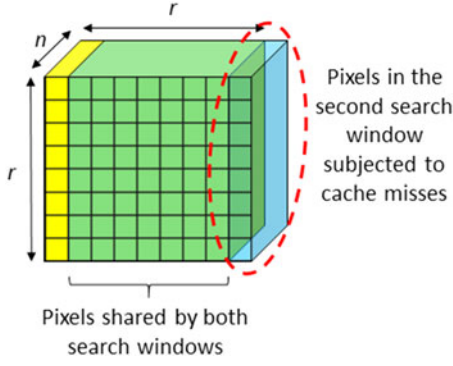


Fig. 2. Depiction of cache misses for a cache that can accommodate at most one whole search window.

$$OI_{\infty} = (p^2 \times r^2 \times \zeta) / \xi$$

With an infinite cache, each pixel is read only one time, i.e., $\xi = p^2 \times n \times 8$ bytes are read from the main memory. Hence,

$$\begin{aligned} OI_{\infty} &= \left(p^2 \times r^2 \left(n \times 20 + \binom{n}{2} \times 49 + 4 \right) \right) / (p^2 \times n \times 8) \\ &= r^2 (n^2 \times 49 - n \times 9 + 8) / (n \times 8) \\ &\approx r^2 (n^2 \times 49) / (n \times 8) = 4 \times n \times r^2 \end{aligned}$$

The above analysis implies that, for an infinite cache, the operational intensity is workload dependent (as a function of the number of input interferograms and the area of the search window), and, importantly, can reach very large values.

Finite, non-zero-cache. Operational intensity for the non-zero-cache scenario,

$$OI_c = (p^2 \times r^2 \times \zeta) / \xi$$

Since, the cache only encounters compulsory and capacity misses, the total number of main-memory references is $\# \text{capacity misses} + \# \text{compulsory misses}$. If all the cache references result in misses, i.e., all the bytes are read from the main-memory then operational intensity will be the same as the zero-cache scenario. If m is the *cache miss rate*, then we can use the following equation to approximate operational intensity for the non-zero cache scenario:

$$OI_c = OI_0 \times (1/m)$$

Next, we compute m which depends on the cache capacity c . We ignore compulsory misses (i.e., we assume cache is warm), as seen in the preceding sections, number of compulsory misses are negligible compared to the total reads.

Cache Miss Rate. Temporal coherence computation involves interferogram generation and computing all the arcs within the predefined search window. This requires reading each pixels within the search window from all n input interferograms. We assume, the interferogram network is generated once the required pixels are brought into the cache, i.e., for k^2 references, there will be no cache misses if all k pixels are already in the cache (i.e., there are enough registers/cache memory available to hold the partial network created on the fly). Total number of cache or memory references within a search window is $n^2 \times r^2$.

Two adjacent pixels' search windows overlap; they share $(n \times r^2) - (n \times r)$ common pixels (Fig. 2). If a cache can accommodate at most the pixels in a fixed sized search window, i.e., $c = n \times r^2$, then a consecutive temporal coherence computation should read $(n \times r^2) - (n \times r)$ pixels from the cache and only $n \times r$ pixels from the main memory. Hence, cache miss rate,

$$m_1 = \# \text{misses} / \# \text{references} = (n \times r) / (n^2 \times r^2) = 1 / (n \times r)$$

Next, we assume the cache is much smaller and can only accommodate about one-fourth of the pixels in the search window, i.e., $c = (n/2) \times (r/2)^2$. In this case, a consecutive temporal coherence computation reads $(n/2) \times (r/2)^2 - (n/2) \times (r/2)$ pixels from the cache and $(n \times r^2) - ((n/2) \times (r/2)^2 - (n/2) \times (r/2))$ or $((3 \times n \times r^2) + (2 \times n \times r)) / 4$ pixels are read from the main memory. Cache miss rate,

$$\begin{aligned} m_2 &= (((3 \times n \times r^2) + (2 \times n \times r)) / 4) / (n^2 \times r^2) \\ &= (3 / (4 \times n)) + 1 / (2 \times n \times r) \gg m_1 \end{aligned}$$

From the preceding analysis, we can see that the operational intensity for the non-zero-cache scenario depends on the cache miss rate, which depends on the size of the cache, and the number of input interferograms and the area of the search window. $OI_c = OI_0 \times (1/m)$ or $OI_c \propto n \times r$ indicates that on a real system, it is possible to for *PtSel* to achieve very high operational intensity. High operational intensity is an indicator of the application being compute bound and performance is most likely bottlenecked by the processing rate of a platform. Later, we empirically demonstrate this is true for *PtSel*.

3 PTSEL DESIGN AND THE IMPLEMENTATION

In this section, we discuss design and implementation of *PtSel* for both the CPU and GPU systems. We also present several optimizations and discuss their tradeoffs.

3.1 A Parallel CPU Implementation

We start from a parallel C++/OpenMP-based CPU implementation that was previously developed at 3vGeomatics. The CPU implementation parallelizes temporal coherence computation across pixels. Each pixel in the image is mapped to a unique OpenMP thread. Each thread (sequentially) computes the temporal coherence of all the arcs within a pixel's search window and identifies the neighbour which maximizes temporal coherence.

It has been observed that, often, highly coherent neighbours are the ones that are closest to the pixel itself. The CPU implementation takes advantage of this property: a thread mapped to a pixel begins with computing coherence of the neighbour closest to the pixel itself and gradually expands the search radius; essentially exploring neighbours along a spiral-path (Fig. 3) which expands outwards from the pixel. Additionally, the implementation takes a 'threshold' (0.0, 1.0]: along the spiral-path, as soon as a neighbour pixel with coherence is greater than or equal to the threshold is found, the search is truncated. Although, the spiral-path approach causes irregular data access, in practice, for most datasets, it works better than exploring

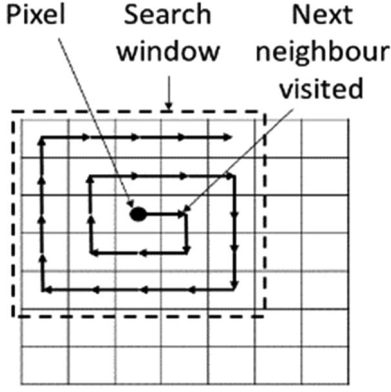


Fig. 3. Visiting pixels within a search window along a spiral-path.

neighbours in a row-major fashion, as it provides work-efficiency by reducing the number of expensive temporal coherence computations.

Patches. Since, for most datasets, it is impossible to load the entire workload in DRAM, the CPU implementation divides the image into *patches* and process one at a time. *PtSel* aims to operate with a large search windows for higher accuracy. The number of common pixels among two adjacent patches depends on the size of the search window. A smaller patch size restricts the size of the search window as well as increases the percentage of redundant pixels among patches. For this reason, *PtSel* aims to maximize the patch size. The memory requirements for a patch depend on its spatial dimension and the size of the interferogram network. One may argue that for the CPU implementation to perform at its best, the patch size should be small enough to fit in the CPU's last level cache (e.g., 40MB on our system). Unfortunately, in practice, this is not even sufficient for a 15×15 pixels search window.

Limitations. The CPU implementation does not exploit vector instructions, e.g., SSE/AVX, that modern CPUs offer. In Section 5, we discuss why we did not consider implementing a vectorized CPU implementations and instead directly went for a GPU implementation.

3.2 Overview of the GPU Implementation

The roofline analysis suggests that *PtSel* will have a high operational intensity even for limited cache sizes, thus it should benefit from the superior FLOP-rate GPUs offer (5-6x higher than that of mainstream CPUs).

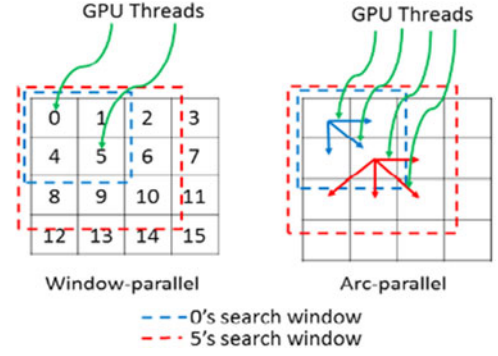


Fig. 5. GPU thread mapping in window and arc-parallel approaches.

PtSel embeds data parallelism at multiple levels: First, for every pixel, within a given search window, finding the neighbour which maximizes temporal arc coherence can be done in parallel. Second, temporal arc coherence computations are independent, so they can be performed in parallel. Our design goal is exposing sufficient parallelism to efficiently harness GPU capabilities. The discussion features a number of design alternatives we have explored.

Fig. 4 presents a high-level overview of the *PtSel* GPU implementation. The user specifies the input dataset to be processed and two parameters: the patch size and the search window size. The input is first divided into patches that are copied to a GPU one at a time. First, the *temporal coherence compute kernel* is responsible for combined interferogram network generation and temporal arc coherence computation. Then, the *max-reduction kernel* performs the reduction operation on the output of the first kernel to find the arc with the maximum coherence. The output of the second kernel is then copied back to the host.

Harnessing Multiple GPUs. *PtSel* can harness multiple GPUs and can process multiple patches simultaneously. The workload is divided into same size chunks (i.e., collections of patches) equal to the number of GPUs. Each GPU processes its own share of the workload independently and coordination between the GPUs is not required.

3.3 Parallel Temporal Coherence Computation

Temporal arc coherence computation is the most time consuming operation (over 90 percent of the total time). This kernel generates on the fly the network consists of $\binom{n+1}{2}$ interferograms from n input interferograms as well as computing arc coherences (as described in Section 2.3).

GPGUs are used efficiently when the application offers massive parallelism (i.e., thousands of threads) that allows hiding memory access latency. We have identified two parallelization approaches for combined interferogram network generation and temporal coherence computation, namely, *window-parallel* and *arc-parallel*.

Window-parallel Approach. In the window-parallel approach, each pixel in a patch is mapped to a GPU thread (Fig. 5). Parallelism is achieved across pixels as each thread generates the interferogram network for the pixels within a search window and computes temporal coherences of all the arcs within the search window.

The access pattern on the input data for spatio-temporal processing, i.e., interferogram network generation and

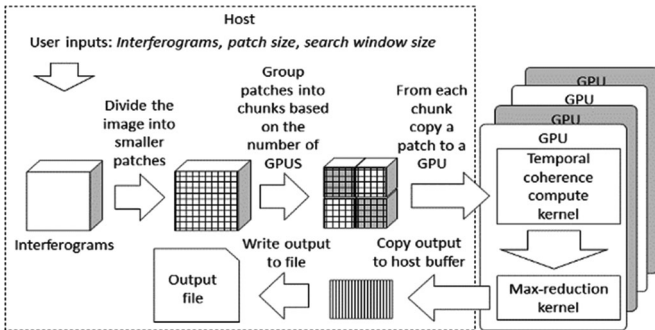


Fig. 4. High-level illustration of the *PtSel* GPU implementation showing tasks on the host and the primary GPU kernels.

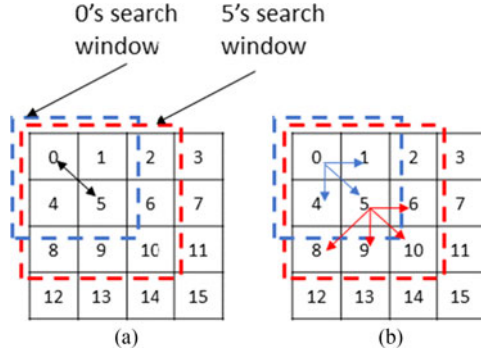


Fig. 6. A 3×3 pixels search window. (a) Shared arc by two search windows. (b) Showing directions from which different arcs are computed according to the half-window technique.

visiting nearby neighbours, resembles that of a 3D array. The resulting strided accesses may cause non-coalesced memory accesses on the GPU.

Limits to Exposed Parallelism. The number of parallel threads the window-parallel approach can exploit is limited by the spatial dimension (i.e., the area) of a patch. Typical workloads (a network consists of thousands of interferograms) lead to up to a roughly 500×500 pixels patch on a GPU with 6 GB memory. For GPUs with a smaller amount of memory or for deeper interferogram stacks, we need a solution that exposes more parallelism.

Arc-parallel Approach. The arc-parallel approach exploits finer granular parallelism at the arc level. As interferogram generation and temporal coherence of any two arcs can be computed independently, the arc-parallel approach, maps each arc to a GPU thread to compute temporal coherence in parallel. We have explored two variants of this technique:

Arc-parallel-A aims to optimize for maximal memory coalescing. In CUDA, requests to the device memory from the 32 threads in a warp can be grouped into a single memory transaction as long as the requested addresses are aligned. Reducing the number of memory transactions means lowering the chance of stalls due to threads waiting for data. Arc-parallel-A leverages this feature of GPUs and exploits a *structure of arrays* (SOA) to store the image data so that a warp mapped to 32 consecutive arcs always accesses 32 consecutive indices in the SOA. This allows eliminating strided accesses to the 3D array at a cost of a larger memory footprint. Although, this approach guarantees memory coalescing, it increases memory footprint and traffic due to introduced data redundancy as the same pixel appears multiple times in the input data source, once for each of its neighbours.

Arc-parallel-B aims to improve locality as each pixel is shared by multiple arcs, equal to $2 \times \text{number of neighbours in a search window} \times \binom{n+1}{2}$ (without the half-window technique). This indicates that temporal coherence computation can benefit from improved locality. The size of the L1 cache on each streaming multiprocessor is 64 KB (shared with the block level shared memory). Kepler GPUs also have larger 1.5 MB L2 cache shared by all the streaming multiprocessors.

In order to improve data locality, arc-parallel-B employs an innovative arc-to-thread mapping scheme to enhance L2 caching: arcs are grouped so that each group forms a (partially) complete graph (K_9 complete graph [20] minus four edges) consisting of 32 edges (Fig. 7c). These 32 arcs are mapped to 32 consecutive threads in a warp.

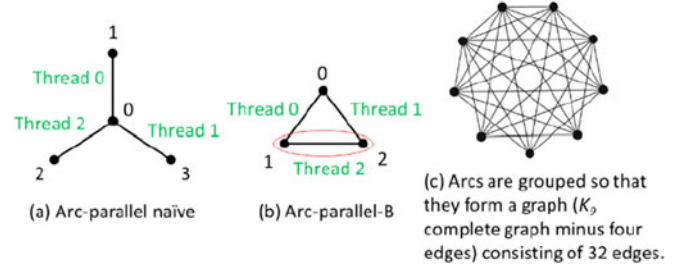


Fig. 7. Arc grouping and arc-to-thread mapping in arc-parallel-B.

Fig. 7. illustrates the idea using three arcs-sharing common pixels. Fig. 7a shows naïve ways of mapping arcs to threads. Fig. 7b shown how threads are mapped to arcs in arc-parallel-B. It is highly likely that Thread 2, mapped to the arc (1, 2), can read all the data, already requested by the other two threads, from the L2 cache. The objective is to reduce stalls at the warp level due to data requests to the device memory through improving cache hit rate.

Common Optimization 1: Eliminating redundant computation. Two adjacent pixels' search windows overlap and they share a common arc (Fig. 6a). In a naïve implementation this arc coherence is computed twice, once for each window. We use an optimization technique, which we dub *half-window*, where we only compute arc coherence for the pixel with the smaller index in an arc.

As a result of this optimization, multiple threads cooperatively compute the arc coherences of all the neighbours of a pixel. It is not possible to determine the maximum temporal arc coherence for a pixel until all the threads in the kernel have completed. Therefore, a barrier is needed before the reduction operation. To implement this synchronization without atomic operations, we use a separate GPU kernel to perform the max-reduction operation.

Common Optimization 2: Hiding memory operation latency by overlapping communication with computation. Processing a patch requires copying patch data from the host to the GPU memory and copying back output. These communications happen over the high latency PCIe-bus. We therefore, resort to CUDA's asynchronous memory copy feature to hide communication latency by overlapping memory operations with computation. Concurrent operations are assigned to separate CUDA streams. For example, assume *temporal_coherence_compute* (K1) kernel operates on a patch k . *max_reduction* (K2) uses output produced by K1. As soon as K1 finishes, k is deleted from the GPU memory and we start copying the next patch ($k+1$) to the GPU and launch K1 concurrently on a separate stream.

4 EVALUATION ENVIRONMENT AND TOOLS

Table 2 shows the two servers we have used for our experiments. On both systems, the CPU code was compiled with gcc-4.8.3 with $-O3$ flag and the GPU code with CUDA 6.5 for compute capability 3.5. All the available CPU threads on both systems were used. Each CUDA kernel is configured to have 512 threads per block, since this configuration showed better performance for typical workloads.

We have used a real-world dataset that is 3,260-pixels wide and 2,680 pixels high and has 60 interferograms; hence, there are $\binom{61}{2}$ or 1,830 pairs in the interferogram

TABLE 2
Systems Used in Experiments

	System A	System B
CPU	2x Intel Xeon E5 2650 v2 (Ivy Bridge) @2.6 GHz	4x Intel Xeon E7 4870 v2 (Ivy Bridge) @2.4 GHz
CPU core count	16 cores (32 threads)	60 cores (120 threads)
CPU last level cache	40 MB L3	140 MB L3
System memory	DDR3 - 256 GB	DDR3 - 1.5 TB
GPU	4x Nvidia Geforce GTX TITAN (GK110)	2x Nvidia Tesla K40c (GK110B)
Multiprocessors (SMX)	14	15
GPU core count	2688 @837 MHz	2880 @745 MHz
GPU thread count	2048/SMX	2048/SMX
GPU last level cache	1.5 MB L2	1.5 MB L2
GPU memory	GDDR5 - 6 GB	GDDR5 - 12 GB

network. The shape of the search window and the patch is a square, so we only refer to their respective widths.

We used *perf* tools [21] and Intel Performance Counter Monitor (PCM) [22] for collecting CPU and *nvprof* [23] for GPU performance monitoring unit (PMU) event measurements.

5 USING THE ROOFLINE MODEL TO UNDERSTAND *PtSel* KERNEL PERFORMANCE

This section focuses on the performance of the *PtSel* kernel while the next section focuses on the performance and quality of the end-to-end solution. We evaluate the performance of *PtSel* kernel on the Superscalar -CPU and the many-core -GPU. Using the well know Roofline Model [13], we determine peak performance on the two platforms. Based on the findings of our analysis, we present a discussion justifying the GPU as the architecture of choice to accelerate *PtSel*. Additionally, we use profiling tools to measure performance influencing properties not captured by the roofline analysis.

5.1 The Roofline Model

The roofline model [13] was originally proposed as a utility to visualize performance bounds of an application on a processing platform. The roofline model establishes relation between peak performance, e.g., floating point operations per second, and the *operational intensity* (OI) of an application: the ratio of its generated work units (FLOP-count in our case) to bytes read from memory (in our case main memory). The model plots on a log-log scale the measured performance against its operational intensity. Peak performance for a platform is computed as:

$$\text{Peak performance (GFLOP/s)} = \min (\text{Peak GFLOP/s of the platform, peak memory bandwidth (GB/s) of the platform} \times \text{operational intensity (FLOPs/bytes)})$$

This equation leads to the roofline-like curve in the figures in this section.

5.2 Roofline Analysis of the CPU Kernel

The System. The two CPU sockets on System A have a combined peak 333 GFLOP/s double precision and 665 GFLOP/s single precision floating point performance as the Intel Ivy

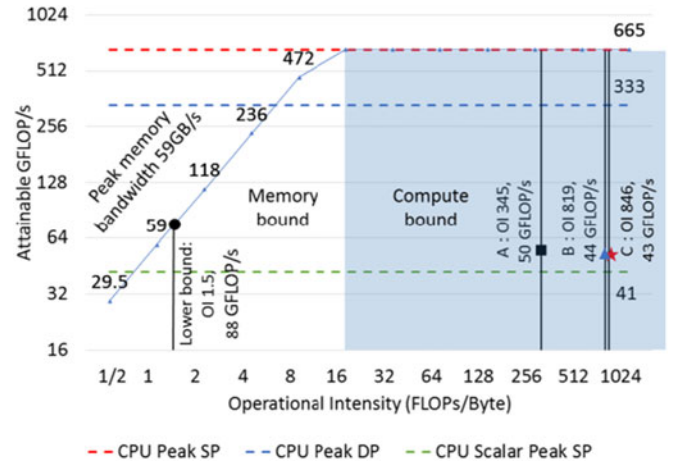


Fig. 8. CPU roofline analysis: on the x-axis is operational intensity and on the y-axis is performance in GFLOPs, both on log scale. The compute bound region has a dark background. The plot shows rooflines for peak single (Peak SP) and double (Peak DP) precision vector operations, and single precision scalar operations (Scalar Peak SP). OI and attainable or sustained performance for the lower bound and three workloads are shown. The bullet points indicate sustained performance for each workload.

Bridge architecture features a 16 lane wide SIMD unit per CPU core and enables up to 8 double or 16 single precision floating point operations per cycle. The peak FLOP-rate of the CPU system is calculated as follows (assuming efficient use of the SIMD units):

$$\text{Peak FLOP/s} = \text{total number of CPU cores} \times \text{frequency of a CPU core} \times \text{maximum floating point instructions per cycle}$$

Tools. We estimate the number of executed floating point operations using the *perf* CPU profiler (using `FP_COMP_OPS_EXE` PMU events). Although PMU events like `MEM_UOPS_RETIRED` and `MEM_LOAD_UOPS_RETIRED` are aimed at measuring ‘load’ and ‘store’ instruction counts, they are not well-suited for measuring off-chip memory activity such as data traffic generated by the application to the DRAM as these counters may include other measurements, like load operations performed as a result of pre-fetching. To this end, we use Intel Performance Counter Monitor which allows measuring data traffic directly at the memory controller.

Roofline Analysis. Fig. 8 is the roofline analysis of the CPU implementation. In Section 2.4, we established that the lower bound (no-cache) operational intensity for *PtSel* is 1.5 FLOPs/byte. This leads to an attainable peak performance of 88.5 GFLOP/s on the roofline plot.

The operational intensity of an application could be workload dependent [13]. Dense Matrix Multiplication and FFT are examples of applications that have such property. Cache efficiency can be influenced by the workload size, e.g., matrix dimension, skewing the operational intensity with changing workload. In Section 2.4, we have shown that with caching, the operational intensity of *PtSel* is workload dependent. To verify this empirically, we have determined operational intensity for three different workloads (Table 3) on the CPU. Fig. 8 shows roofline plot for three different workloads. The workloads differ across interferogram network size and search window size. The experiment considers a single patch of width 500 pixels instead of the entire image.

TABLE 3
Workloads Used for CPU Roofline Analysis

Workload	A	B	C
Interferogram network size	$\binom{61}{2}$	$\binom{61}{2}$	$\binom{61}{2}$
Search window width (pixels)	51	51	15

Workload *A* has a larger or equal size interferogram network and search window compared to workloads *B* and *C*. For workload *A*, computation load along both spatial and temporal dimensions are high. As a result, higher number of strided accesses are required for spatio-temporal processing. Hence, workload *A* is the least successful in utilizing on-die caches as indicated by the low operational intensity (345) it achieves. Although workloads *B* and *C* differ in terms of the size of the interferogram network and search window, both achieve similar operational intensity, 819 and 846 respectively. Due to smaller interferogram network and search window size respectively, both *B* and *C* issue a lot fewer load requests to the memory subsystem. Additionally, biased compute load along either spatial or temporal dimension results in less strided accesses compared to workload *A*, locality is preserved and the caches are better utilized.

This empirical evidence validates our analytical model in Section 2.4, where we showed that on a real system, operational intensity for *PtSel* can be very high. The model correctly predicts that operation intensity is workload dependent. Also, it can determine bounds for the operational intensity: on a real system, operational intensity for the three different workloads fall between the two *non-zero-cache* scenarios discussed in Section 2.4, enduring low and high miss rates respectively (see Table 4).

Average sustained processing rate on the CPU is 45 GFLOP/s. Sustained performance is lower than what is attainable (88 GFLOP/s) for the lower bound OI as suggested by the roofline plot. Given that the application code considers only *scalar* operations, this is not surprising. Ivy Bridge CPUs can do one floating point operation per cycle. The peak for scalar only floating point operations is about 41 GFLOPs. Below we explain why sustained performance is slightly higher than the peak scalar performance.

Explaining Performance Bounds. Performance close to the peak is only attainable when the SIMD instructions are properly utilized. Application code relying on scalar instructions can only offer limited throughput (16x lower than SIMD). Efficient use of SIMD instructions however is challenging. The learning curve is steep: lack of high level APIs and wrappers require low level understanding of the instruction set.

Profiling shows that, by default, the implementation of gcc math library performs some vector (packed double) operations on our system. The application code performs

TABLE 4
Operational Intensities Predicted by the Analytical Model for Workload A

OI_0	OI_∞	$OI_c (m_1 = 3.3e-4)$	$OI_c (m_2 = 0.013)$
1.5	624,240	4,590	118.4

Cache Miss Rates for the Non-Zero-Cache Scenarios in Section 2.4 are Shown Too

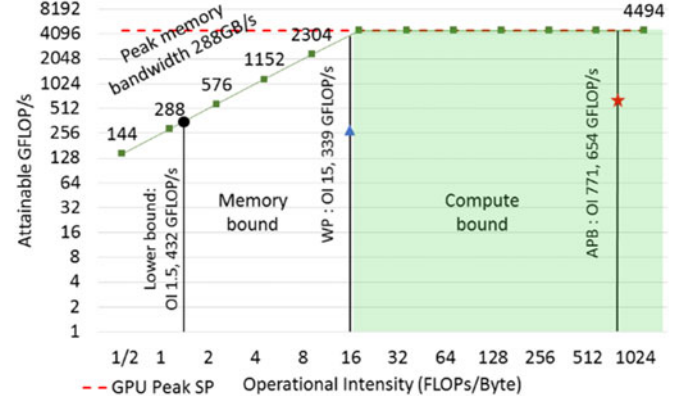


Fig. 9. GPU roofline analysis: on the x-axis is operational intensity and on the y-axis is performance in GFLOP/s. Attainable and sustained performance for lower bound and observed operational intensity are shown separately.

only single precision scalar operations. Hence, the compiled code is a mixture of single and double precision floating point instructions and executes both scalar and vector floating point operations at run time. This explains why sustained performance (FLOP-rate) is slightly higher than peak single precision scalar floating point performance of the platform.

Apart from instruction level parallelism, there are many other influencing factors that heavily impact performance of an application on a CPU system. Irregular data accesses often cause inefficient hardware perfecting which in turn cause cache pollution, negatively effecting performance of a cache sensitive application. Inefficiency of out-of-order execution, for example due to excessive branch miss predictions, causes pipeline stalls leading to dwindled instruction throughput.

5.3 Roofline Analysis of the GPU Kernel

The System. Peak single precision floating point performance of a GPU on System A is 4.5TFLOP/s. As the GPU supports fused multiply-add (FMA) instructions, peak FLOP-rate of a GPU is calculated as follows:

$$\text{Peak FLOP/s} = \text{total number of GPU cores} \times \text{frequency of a GPU core} \times 2.$$

Tools. We measured actual number of floating point operations and bytes read from GPU device memory using the nvprof GPU profiler (using `flop_count_sp`, and `dram_read_throughput` PMU events respectively).

Roofline Analysis. Fig. 9 shows roofline plots for window-parallel (WP) and arc-parallel-B (APB) GPU implementations. The workloads used for CPU analysis cause the GPU PMU counters to overflow. Hence, we use a smaller the workload: there are 15 input interferograms, patch width is 300 and search window width is 21 pixels.

Without optimizations for locality, we expect *PtSel* to achieve a lower operational intensity on the GPU as it has a much smaller last level cache. Achieved OI for WP and APB are 15 and 771 respectively.

Although arc-parallel-A (APA) ensures 100 percent coalesced memory accesses, the data redundancy significantly increases memory traffic on the device. This overshadows the advantage of additional parallelism and aligned memory accesses and, unfortunately, does not improve performance over WP. Therefore, we do not discuss APA in the rest of the paper.

When the size of the patch is small, WP lacks the massive parallelism of the arc-parallel approach. This is the likely explanation for the limited sustained performance observed: the roofline model suggests that for an OI of 15 the application is compute-bound, however the sustained performance is only 339GFLOPs.

APB's design aims to harness superior parallelism and take advantage of the better data locality. Performance numbers clearly demonstrate the design choices made by APB were in the right direction. The arc ordering that APB follows, significantly improves data locality, hence caching on the GPU and therefore, reduce data traffic to the device memory. L2 cache hit rate is 42 percent better compared to that of WP. Sustained performance by APB is 654 GFLOP/s.

PtSel GPU implementations take advantage of fused multiply-add (FMA) instructions available on modern GPUs. In the case of APB, about half of the total floating point instructions are FMA instructions and for WP, one third of the total floating instructions are FMA instructions.

Explaining Observed Performance Bounds. Many of the performance influencing properties are not captured by the roofline plot, therefore, we collect per kernel GPU PMU event statistics to better explain performance.

Nvidia GK110 architecture allows a single thread to use up to 255 registers and 16 concurrent thread-blocks per-SMX. Given the fixed number of registers per-SMX, high per-thread register usage condenses the number of concurrent thread-blocks and negatively effects overall parallelism and application throughput.

Profiling of the *temporal coherence compute kernel* shows that *PtSel* incurs high register usage, 38 registers per thread. This means for a 512 threads per block configuration, $512 \times 38 = 19,456$ registers will be used by each block. Each multiprocessor has 65,536 registers meaning at most three concurrent thread-blocks per multiprocessor. This is one of reasons why sustained performance of *PtSel* on the GPU is far from the platform's peak.

Note that it is not surprising that the CPU and the GPU solutions are far from reaching respective platform's peak; sustained performance of most real-world applications is usually no more than 20 percent of a platform's peak [24].

5.4 Justifying the Use of GPUs to Accelerate *PtSel*

In this section, we argue *why we opted for a GPU implementation without attempting a SIMD implementation for the CPU.*

Performance. In theory, a SIMD CPU implementation could offer at most 16x speedup (with respect to FLOP-rate) over a scalar implementation. However, linear speedup with SIMD width is rarely observed in practice. More importantly, according to the roofline plot, even the lower bound of operational intensity (assuming no caching), on a GPU yields 432 GFLOP/s attainable performance; within 65 percent of the CPU peak of 665 GFLOP/s.

Programmability. Unlike SIMD on GPUs, SIMD does not execute instructions in parallel if a divergent branch exists, and the execution of the entire unit is serialized. SIMD has strict data alignment and branch condition requirements. Furthermore, we found that in order to satisfy boundary conditions and corner cases, for the SIMD to work efficiently, the implementation would require developing multiple stencil operators. This is much more challenging than

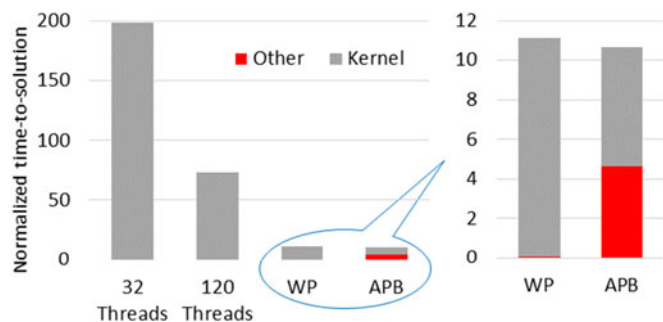


Fig. 10. Comparison of time-to-solution: GPU results are shown separately for better visibility (right). 'Other' refers to time spent in operations other than core *PtSel* kernel execution. 'k Threads' refers to the number of CPU threads used.

developing code for SIMD which is better at tolerating divergent branches and unaligned data and still perform parallel execution. OpenMP's MIMD execution model exposes flexible interface for fast development for multi-core processors. However, extracting maximum performance requires good knowledge of the NUMA architecture commonly adopted by modern multi-core processors. In our experience NUMA-aware programming is more challenging than writing CUDA code.

These are the reasons behind our option for a GPU implementation instead of attempting a SIMD and NUMA-aware CPU implementation. Furthermore, as it is possible to achieve linear speedup with increasing GPU count (see Section 6.2), adding more GPUs to a system provides a cheaper alternative to scale capacity than assembling a new system with more CPU sockets.

6 END-TO-END EVALUATION

We evaluate the end-to-end performance and the quality of the solution proposed. On the one side we compare CPU and GPU implementations with respect to time-to-solution, scalability and energy efficiency. On the other side we compare *PtSel* with the widely-used PSI software package StaMPS and highlight the quality of the solution offered in terms of point section accuracy.

Unless otherwise specified, an experiment was performed on System A, the workload has all 60 interferograms and arc-parallel-B GPU implementation was used. All the reported GPU time-to-solution include time to copy data from the host to the GPU and copying back output to the host memory.

6.1 Comparison of Time-to-Solution

The original objective of this project was to improve the time-to-solution of *PtSel* so it can be used in the production settings. Fig. 10 depicts time-to-solution for the CPU and the GPU solutions. Here, the search window and patch width are 51 and 400 pixels respectively. CPU experiments were performed both on System A and B using all the available CPU threads, while the GPU results are for a single GPU on System A.

The performance advantage of the arc-parallel-B GPU kernel is attributed to creating group of arcs following the pattern described earlier. Unfortunately, this step is not cost-free and is currently this is performed on the CPU as parallelizing this step is difficult. Fig. 10 shows a

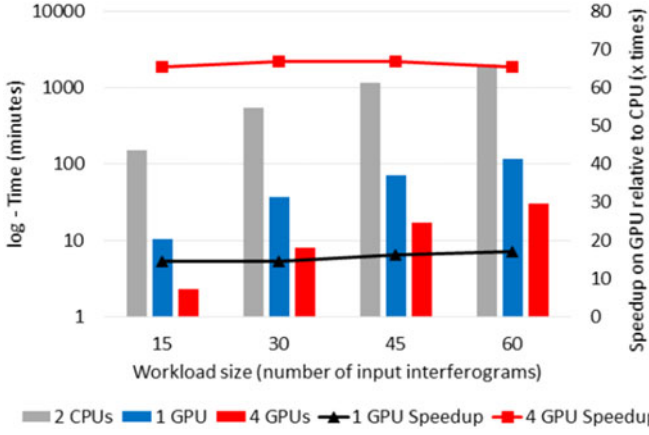


Fig. 11. CPU versus GPU time-to-solution for different number of input interferograms (left y-axis). Speedup on GPU over CPU on the right y-axis.

breakdown of time-to-solution. The figure separates kernel execution time and the other operations, e.g., memory copy between the host and the accelerator and kernel preprocessing. The time spent in memory copy operations is the same for all the kernels and negligible. (about half a second where kernel execution time is in the order of minutes). As we can see, the arc-parallel-B only spends a little over half of the total time in executing the GPU kernel while rest of the time is spent in preprocessing on the host, i.e., re-ordering the arcs. Window-parallel is free of such preprocessing overhead and as a result of which (for the given parameters) arc-parallel-B is only 1.03x faster with respect time-to-solution. Both the GPU solutions are about 18x and faster than the CPU solution on the dual-CPU system and about 6.7x faster than the CPU solution on the quad-CPU system.

Additionally, the arc-parallel technique also has a larger memory footprint. For the largest workload, on System A, arc-parallel allows us to have patches with maximum width 400 pixels, whereas window-parallel can accommodate a patch with width 500 pixels. The next section studies the impact of patch size on performance.

6.2 Scalability with Respect to Workloads and Application Parameters

Number of Interferograms. Fig. 11 compares time-to-solution of the CPU and the GPU implementation for different workload sizes (i.e., varying number of input interferograms). Here, the search window and patch width are fixed to 51 and 400 pixels respectively. On System A, the CPU implementation takes about 33 hours to process the largest workload, whereas the GPU implementation takes about 1 hour and 55 minutes on a single GPU and 31 minutes on four GPUs; offering about 18x and 65x speedup respectively. On average, the CPU implementation takes 33 minutes to process a single patch, whereas using four GPUs, four patches are processed in about two minutes. The GPU implementation scales almost linearly with increasing workload and GPU count.

Note that, unlike the GPU implementation, the CPU implementation actually does not compute all the arcs throughout the interferogram network: a thread stops computing further arcs if it has already found one whose coherence is greater than or equal to the threshold (0.71). Without this optimization, the CPU implementation performs a lot worse.

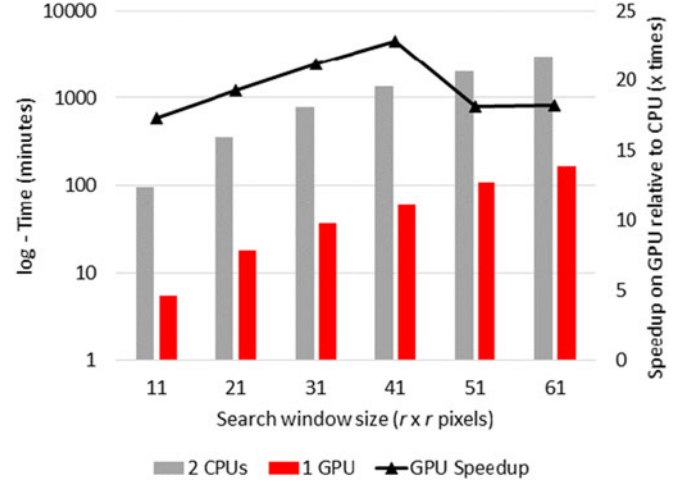


Fig. 12. CPU versus GPU time-to-solution for different search window sizes (left y-axis). Speedup on GPU over CPU on the right y-axis.

Search Window Size. In Section 2.3, we discussed the significance of the search window size: this allows a pixel to consider neighbours at a greater distance at the cost of increased number of arc coherence computations. Fig. 12 compares the time-to-solution of the CPU and GPU implementations for different search window sizes. Here, the patch width is fixed to 400 pixels. For smaller window sizes, speedup over CPU steadily increases but it drops slightly at width 51 and plateaus beyond that. A plausible explanation of this behavior is the following: Threads created by the arc-parallel approach increases with increasing patch and search window size. At the physical level, a GPU can only run a fixed number of threads simultaneously. Threads that are not scheduled to run in parallel are scheduled sequentially. Hence, excessive thread creation might not yield any advantage. For the largest search window, the GPU implementation is about 18x faster than the CPU implementation. For the smallest search window, the GPU have the least advantage. This is because with larger windows, in addition to increased computation the advantage of the large CPU cache starts to diminish and the frequency of request to the slower main memory increases.

Patch Size. Next, we study influence of the patch size on time-to-solution. A smaller patch size has a number of drawbacks: It causes increased number of patches; this in turn elevates the frequency of copying data back and forth between the host and the GPU. Since, adjacent patches partially overlap and share common pixels along the patch boundary, it also increases the amount of data redundantly copied on to the GPU memory. Furthermore, decreasing the patch size also decreases the number of GPU threads that would be used to process the patch which may underutilize available parallelism.

Fig. 13 compares time-to-solution for different patch sizes on System B which has enough GPU memory to accommodate a patch that is 600 pixels wide. Here, the search window width is fixed to 51 pixels. As we can see, for window-parallel, performance improves with increasing patch size. In addition to reduced frequency of communication between the host and the GPU, window-parallel benefits from higher number of threads being exploited by a larger patch. Results for arc-parallel, however, are rather

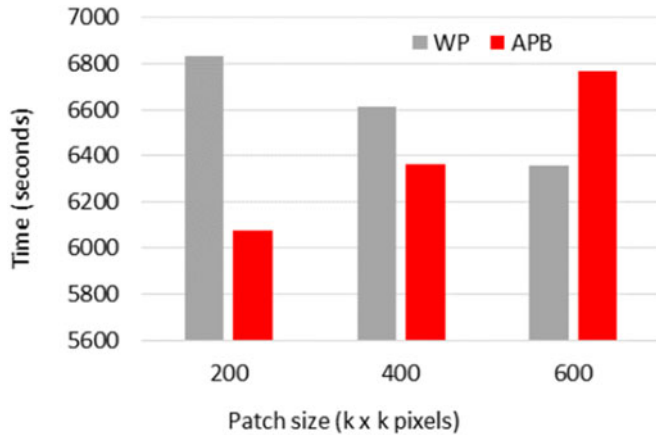


Fig. 13. Comparison of time-to-solution of the GPU solutions for different patch sizes.

interesting. For the chosen parameters, performance decreases with increasing patch size. Following explains this behavior: With a very large search window and a large patch size, the number of threads arc-parallel deploys may increase significantly and a GPU can only execute a fixed number of physical threads simultaneously. The overhead of managing excessive number of threads surpasses the advantage of arc-parallel's ability to exploit more threads. However, performance of arc-parallel-B will increase on the future GPUs that promise more parallelism at the hardware level (more physical cores or threads).

If the entire workload fits in the main memory, for the CPU solution, it is possible to process the image without creating patches. On System B it takes about 11 hours to process the workload in Section 6.1 without creating patches, an hour less compared to creating 400×400 pixels patches.

6.3 CPU versus GPU: Energy Profile

We measure energy consumption of the CPU and GPU implementations on System A, for the largest workload as in Section 6.2. Power (watts) is measured at the wall outlet using a WattsUP meter which collects samples at one second intervals [25]. Thermal design power ratings for the CPU is 95 W, the GPU is 250 W and loaded power rating for the DDR3-1333/1600 256 GB memory is about 80 W. Idle system power is about 280 W (including GPU idle power which is about 20 W per GPU). Average power consumption by the CPU and the GPU solutions are 401 W and 893 W respectively. Given the time they require to reach completion, power usage by the CPU implementation is 13.23 kilowatt hour. When using four GPUs, the window-parallel GPU implementation measures to only 0.46 kilowatt hour, a 28x improvement over the CPU implementation.

TABLE 5
SAR Datasets Used for Comparison with Stamps

Dataset	Satellite orbit path	Track	#Interferograms	Dimension (pixels)
Los Angeles (urban)	Ascending	392	31	2,500×15,000
San Joaquin (rural)	Descending	485	67	3,000×15,000

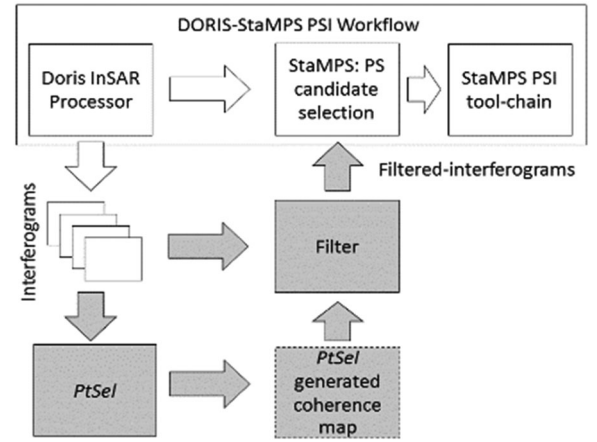


Fig. 14. High-level overview of DORIS-StaMPS workflow and how *PtSel* was incorporated to the end-to-end processing chain.

6.4 Verifying Point Selection Quality

Methodology and Experiments. In order to verify the PS pixel selection quality of *PtSel*, we used StaMPS [14]—a widely-used open-source software for PSI analysis. StaMPS PSI technique is based on [12]. While there exist other sophisticated PS-InSAR techniques such as Small Baselines [29] and SqueeSAR [30] which could provide a baseline for comparison, their applicability to large datasets are limited by slow processing.

We compare PSI results produced by StaMPS with and without introducing *PtSel* to the standard processing chain. Two ENVISAT ASAR datasets representing two different environments were considered for the evaluation: 1. An *urban* area over the City of Los Angeles and surroundings, and 2. The *rural* vicinity of the San Joaquin area in California consists of several mines, mountains and a lot of vegetation (Table 5). We perform end-to-end processing for each dataset, i.e., from interferogram creation to producing the final deformation map indicating average linear deformation velocity (mm/year).

The interferograms are created using DORIS InSAR processor [26]. We used the ENVISAT Precise Orbits from DEOS [27] and the NED/DEM from the US Geological Survey (USGS) [28].

To verify effects of *PtSel* processing, we use *PtSel* generated coherence map to spatially filter the interferograms. The filter replaces each pixel data (a phasor) in each of the original interferograms by the average of the phasors of its neighbouring pixels (in the same interferogram), where the phasor of a neighbour is weighted by its coherence found in the *PtSel* generated coherence map. The filter considers the neighbours in the same distance as used by *PtSel* to generate the coherence map. The filtered interferograms are used as the input to the StaMPS's PS candidate selection module and the rest of StaMPS's standard PSI procedures are applied to obtain the final deformation map (Fig. 14). We produced *PtSel* coherence maps for two different search window sizes, 11×11 and 35×35 pixels, for each of the datasets in Table 5.

Results and Discussion. We have obtained three sets of results for each dataset: Directly applying StaMPS PSI without using *PtSel*; and using *PtSel* with a 11×11 and, respectively, 35×35 pixels search window.

Table 6 summarizes the results for the following metrics: the number of PS identified, linear deformation detected (mm/

TABLE 6
Result Summary—Number of PS Identified, and
Linear Deformation Detected: Maximum Up-Lift and
Subsidence for Three Different Experiments

Dataset		Standard StaMPS	<i>PtSel</i> + StaMPS - 11x11 pix. s/w	<i>PtSel</i> + StaMPS - 35x35 pix. s/w
Los Angeles	# PS identified	374,095	626,982	724,320
	Max. up-lift	10.96	11.05	10.66
	Max. subsid.	10.96	11.11	10.69
San Joaquin	# PS identified	212,741	286,776	323,686
	Max. up-lift	8.7	10.8	7.97
	Max subsid.	28.15	33.2	28.38

year): maximum uplift and subsidence. The visual line-of-sight linear deformation velocity for both datasets these all three experiments are shown in Figs. 15 and 16 and respectively. We observe the following:

- Importantly *PtSel* increases the number of final PS in both urban and rural datasets: about 2x for Los Angeles and 1.5x in the case of San Joaquin.
- PtSel* offers superior coverage in areas where the original method offered no coverage. *PtSel* was able unearth points with varying deformation signals in wide areas ignored by standard StaMPS processing (notice the areas inside the red boxes in Figs. 15 and 16).

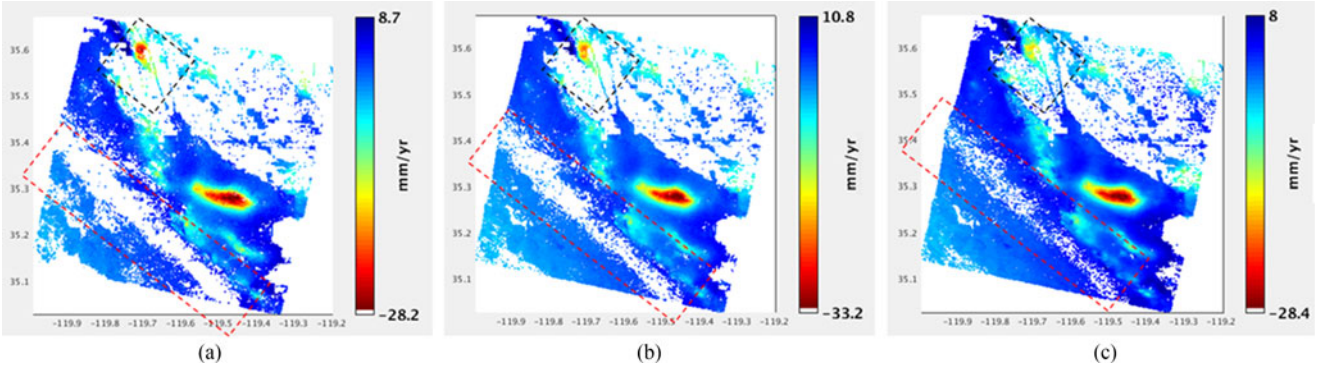


Fig. 15. Line-of-sight linear deformation velocity (mm/year) for the San Joaquin dataset. A non-zero positive value indicates up-lift and negative value indicates subsidence. (a) Results obtained following standard StaMPS PSI. (b) Results for *PtSel* augmented StaMPS PSI. A 11×11 pixels neighbour search window was used by *PtSel* to generate the coherence map. (c) Same as (b), except that a 35×35 pixels neighbour search window was used.

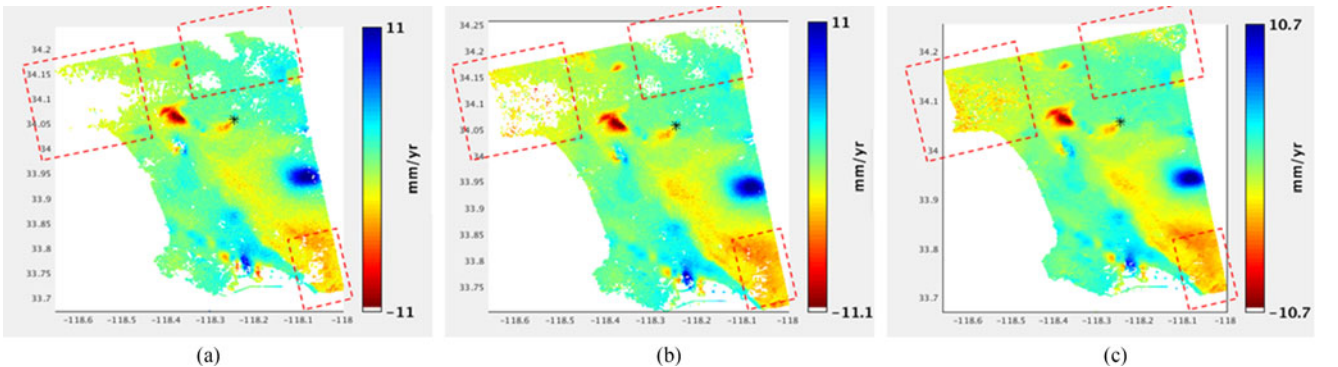


Fig. 16. Line-of-sight linear deformation velocity (mm/year) for the Los Angeles dataset. A non-zero positive value indicates up-lift and negative value indicates subsidence. (a) Results obtained following standard StaMPS PSI. (b) Results for *PtSel* augmented StaMPS PSI using an 11×11 pixels neighbour search window to generate the coherence map. (c) Same as (b) but using a 35×35 pixels neighbour search window.

TABLE 7
Common Points Identified by StaMPS and *PtSel*

	<i>PtSel</i> 11x11 pix. s/w	<i>PtSel</i> 35x35 pix. s/w
Los Angeles	66.05%	70.31%
San Joaquin	75.37%	79.95%

- The larger the neighbour search window, the smoother the detected deformation signal is. A smaller search window, on the other hand, is better at preserving the deformation patterns effecting smaller extents (the areas within the black boxes in Fig. 15).
- Liner deformation velocity measurements are mostly comparable except for the *PtSel* 11×11 pixels search window for the San Joaquin dataset, which is showing higher deformation rate (Fig. 15b). Given that there exists no ground truth, more analysis is required to understand the difference in results, a candidate for our future work with our collaborators.

Table 7 lists statistics on the common points identified by the standard StaMPS PSI and the *PtSel* augmented workflow. *PtSel* selects at least 66 percent of the points identified by standard StaMPS, and up to ~80 percent for the rural dataset. This boosts our confidence in *PtSel*'s point selection ability. One interesting question relates to identifying the points eliminated, possibly incorrectly, as a result of including *PtSel*. One of the objectives of *PtSel* is to identify points in the presence of distributed scatterers,

prevalent in rural and high-noise environments. *PtSel*'s collective filtering approach, which considers similar nearby neighbours, helps to identify useful points in the presence of distributed scatterers. However, one shortcoming of this approach is that isolated PS, different from all other points within the search window, may get filtered out, with non-PS points essentially eliminating them from the PS list. Such isolated PS could be selected by StaMPS but not by *PtSel* for being with neighbours which were selected instead.

In Step 3 and Step 4 of StaAMP PSI (see StaMPS manual 3.3.b1, Sections 6.3 and 6.4 [14]), pixels are rejected based on coherence value and neighbours are weeded to remove *side-lobes* [17]. We believe these steps also contribute to not having all the points identified by standard StaMPS, in the deformation map produced by *PtSel* augmented StaMPS.

7 SUMMARY

This paper presents a new algorithm, *PtSel*, for the persistent scatterer selection problem. The proposed solution is free of model assumptions as required by most existing techniques. *PtSel* aims for acceleration both in terms of time-to-solution and PS coverage, and its design choices were conceived accordingly.

Motivated by the parallelism it exposes, we have implemented *PtSel* for GPUs. The GPU implementation not only offers superior performance but also scales linearly with increasing GPU count and workload size. On a single consumer grade GPU, it offers 18x speedup over a parallel CPU implementation, while four GPUs offer 65x speedup. We also present a data-to-thread mapping technique which is effective for deep interferogram networks. Although the power rating of the CPU is half of the GPU, GPUs deliver 28x better energy usage.

Comparison with the popular StaMPS package highlights *PtSel*'s ability to increase the spatial PS coverage while keeping the precision of the final result comparable. This is critical in natural environments, where vegetation rapidly decorrelates SAR signal. Volcanoes and mines are found in rural terrains which *PtSel* can help to monitor with higher precision by increasing the number of the resulting PS points. *PtSel*'s current design sacrifices isolated targets with unique models, distinct from all their neighbours. However, these types of points are rarity in the majority of the landscapes and environments we have observed in practice.

REFERENCES

- [1] P. Rosen, S. Hensley, I. Joughin, F. K. Li, S. Madsen, E. Rodriguez, and R. M. Goldstein, "Synthetic aperture radar interferometry," *Pro. IEEE*, vol. 88, no. 3, pp. 333–382, Mar. 2000.
- [2] ESA, "InSAR principles," Dec. 16, 2014. [Online]. Available: http://www.esa.int/About_Us/ESA_Publications/InSAR_Principles_Guidelines_for_SAR_Interferometry_Processing_and_Interpretation_br_ESA_TM-19
- [3] K. Goel, F. R. Gonzalez, N. Adam, J. Duro, and M. Gaset, "Thermal dilation monitoring of complex urban infrastructure using high resolution SAR data," in *Proc. IEEE Geoscience Remote Sensing Symp.*, 2014, pp. 954–957.
- [4] M. Przylucka, G. Herrera, M. Graniczny, D. Co-lombo, and M. Béjar-Pizarro, "Combination of conventional and advanced DInSAR to monitor very fast mining subsidence," *Remote Sens.*, vol. 7, pp. 5300–5328, 2015.
- [5] P. Fokker, B. Wassing, F. Van Leijen, R. Hanssen, and D. Nieuwland, "Application of an en-semble smoother with multiple data assimilation to the Bergermeer," *Geo-Mechanics Energy Environment*, vol. 5, pp. 16–28, 2016.
- [6] V. Singhroy, R. Couture, P.-J. Alasset, and V. Poncos, "InSAR monitoring of landslides on permafrost terrain in Canada," in *Proc. IEEE Int. Geoscience Remote Sensing Symp.*, 2007, pp. 2451–2454.
- [7] A. Hooper, D. Bekaert, K. Spaans, and M. Arkan, "Recent advances in SAR interferometry time series analysis for measuring crustal deformation," *Tectonophysics*, vol. 514, no. 1, pp. 1–13, 2012.
- [8] M. Richards, "A beginners guide to interferometric SAR concepts and signal processing," *IEEE Aerospace Electron. Syst. Mag.*, vol. 22, no. 9, pt. 2, pp. 5–29, Sep. 2007.
- [9] C. W. a. Z. H. A. Chen, "Two-dimensional phase unwrapping with use of statistical models for cost functions in nonlinear optimization," *J. Opt. Soc. Amer. A*, vol. 18, no. 2, pp. 338–351, 2001.
- [10] A. Z. H. Hooper, "Phase unwrapping in three dimensions with application to InSAR time series," *J. Optical Soc. America A*, vol. 24, no. 9, pp. 2737–2747, Sep. 2007.
- [11] P. A. Rosen, S. Hensley, H. A. Zebker, F. H. Webb and E. J. Fielding, "Surface deformation and coherence measurements of Kilauea volcano," *J. Geophys. Res.-Planets*, vol. 101, no. 10, pp. 109–125, 1996.
- [12] A. Hooper, H. Zebker, P. Segall, and B. Kampes, "A new method for measuring deformation on volcanoes and other natural terrains using InSAR persistent scatterers," *Geophysical Res. Lett.*, vol. 31, no. 23, Dec. 2004, Art. no. 5.
- [13] A. W. A. D. P. S. Williams, "Roofline: an insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009.
- [14] A. Hooper, "STAMPS-MTI," Sep. 12, 2013. [Online]. Available: <http://homepages.sse.leeds.ac.uk/~earahoo/stamps/>
- [15] J. Sousa, A. Hooper, R. Hanssen, L. Bastos, and M. Ruiz, "Persistent Scatterer InSAR: A comparison of methodologies based on a model of temporal deformation versus spatial correlation selection criteria," *Remote Sensing Environment*, vol. 115, no. 10, pp. 2652–2663, Oct. 2011.
- [16] A. Ferretti, C. Prati, and F. Rocca, "Permanent scatterers in SAR interferometry," *IEEE Trans. Geoscience Remote Sensing*, vol. 39, no. 1, pp. 8–20, Jan. 2001.
- [17] J. Guo, Y. Xu and L. Fu, "An extended chirp scaling algorithm for spaceborne sliding spotlight synthetic aperture radar imaging," *Chinese J. Aeronautics*, vol. 27, no. 4, pp. 892–902, Aug. 2014.
- [18] G. D. Smith, *Numerical Solution of Partial Differential*. 3rd ed. Oxford, U.K.: Oxford University Press, 1985.
- [19] M. S. A. Hill, "Evaluating associativity in CPU caches," *IEEE Trans. Comput.*, vol. 30, no. 12, pp. 1612–1630, Dec. 1989.
- [20] A. Bondy and U. Murty, *Graph Theory (Graduate Texts in Mathematics)*, London, U.K.: Springer, 2007.
- [21] Perf tools. [Online]. Available: http://perf.wiki.kernel.org/index.php/Main_Page
- [22] Intel performance counter monitor. [Online]. Available: <http://software.intel.com/en-us/articles/intel-performance-counter-monitor>
- [23] Nvidia, "CUDA toolkit documentation," Dec. 16, 2014. [Online]. Available: <http://docs.nvidia.com/cuda/index.html>
- [24] T. Sterling, et al., "Multi-core for HPC: Breakthrough or breakdown?," in *Proc. ACM/IEEE Conf. Supercomputing*, 2006, Art. no. 73.
- [25] WattsUP Meter, Dec. 16, 2014. [Online]. Available: <http://www.wattsupmeters.com>
- [26] B. Kampes, R. Hanssen and Z. Perski, "Radar In-terferometry with public domain tools," presented at the Fringe Workshop, Frascati, Italy, 2003.
- [27] E. Doornbos, R. Scharroo, H. Klinkrad, R. Zandbergen, and B. Fritsche, "Improved modelling of surface forces in the orbit determination of ERS and ENVISAT," *Canadian J. Remote Sensing*, vol. 28, no. 4, pp. 535–543, Aug. 2002.
- [28] D. Gesch, M. Oimoen, S. Greenlee, C. Nelson, M. Steuck, and D. Tyler, "The national elevation dataset," *Photogrammetric Eng. Remote Sensing*, vol. 68, no. 1, pp. 5–32, 2002.
- [29] A. Hooper, "A multi-temporal InSAR method incorporating both persistent scatterer and small base-line approaches," *Geophysical Res. Lett.*, vol. 35, no. 16, Aug. 2008.
- [30] A. Ferretti, A. Fumagalli, F. Novali, C. Prati, F. Rocca, and A. Rucci, "A new algorithm for processing interferometric data-stacks: SqueeSAR," *IEEE Trans. Geoscience Remote Sensing*, vol. 49, no. 9, pp. 3460–3470, 2011.



Tahsin Reza received the BSc and MSc degrees from Memorial University of Newfoundland and Carleton University in Ottawa, respectively. He is working toward the PhD degree in computer engineering at University of British Columbia. His research interests are parallel and distributed systems and HPC applications. His professional experiences include working at Lawrence Livermore National Lab and BlackBerry Ltd. in Waterloo, Ontario.



Aaron Zimmer received the BASc degree in engineering physics from University of British Columbia. He is currently a software engineer with 3vGeomatics. He focuses on InSAR filtering algorithms and local co-registration of SAR images by means of speckle tracking. The code that he has developed is used in production to process large InSAR data sets and is written to make the most use of available computing resources.



José Manuel Delgado Blasco received the MSc degree in electrical and electronic engineering from Polytechnical University of Valencia, in 2010. From 2010 to 2014, he worked as a PhD researcher with Delft University of Technology (The Netherlands) and KU Leuven (Belgium) on Earth Observation applications. In 2014, he joined the Progressive Systems team to work in ESA Research and Service Support (RSS) at the European Space Research Institute (ESA-ESRIN) in Frascati, Italy as an Earth Observation Research

Engineer. He is also collaborating as a volunteer researcher with the Grupo de investigación Microgeodesia Jaén, Universidad de Jaén, located in Jaén, Spain. His research interests include EO applications, focusing on landscape dynamics, data fusion and SAR interferometry.



Parwant Ghuman, received the BASc degree in electrical engineering from University of British Columbia, is the CTO with 3vGeomatics, where he focuses on developing image processing technology for maximizing information extraction from SAR data stacks. His research focuses on automated monitoring solutions for difficult datasets exhibiting noise and under-sampling across diverse applications including resource extraction, civil infrastructure, and permafrost. He is also interested in scalability challenges associated with efficient processing of large volumes of SAR data.



Tanuj Kr Aasawat is working toward the graduate degree in computer engineering at University of British Columbia. He is working on large scale graph processing on hybrid CPU-GPU systems. He interned at IBM Almaden Research Center where he contributed to building Deep Learning support using GPUs for Apache SystemML, a large-scale declarative Machine Learning framework by IBM.



Matei Ripeanu received the PhD degree in computer science from The University of Chicago. After a brief visiting period with Argonne National Laboratory, Matei joined the Electrical and Computer Engineering Department of the University of British Columbia. He is broadly interested in distributed systems with a focus on self-organization and decentralized control in large-scale Grid and peer-to-peer systems.

► For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.