Shen Liang[1], Antoine Lucas[2], and Themis Palpanas[3]

[1]Data Intelligence Institute of Paris (diiP), Université Paris Cité Paris
[2]Université Paris Cité, CNRS Paris
[3]Institut de Physique du Globe de Paris (IPGP), LIPADE, Université Paris Cité & French University Institute (IUF) Paris

January 17, 2023

POPS: An Efficient Framework for GPU-based Feature Extraction of Massive Gridded Planetary LiDAR Data [Scalable Data Science]

Shen Liang
Data Intelligence Institute of Paris (diiP), Université Paris Cité
Paris, France
shen.liang@u-paris.fr

Themis Palpanas
LIPADE, Université Paris Cité & French University Institute (IUF)
Paris, France
themis@mi.parisdescartes.fr

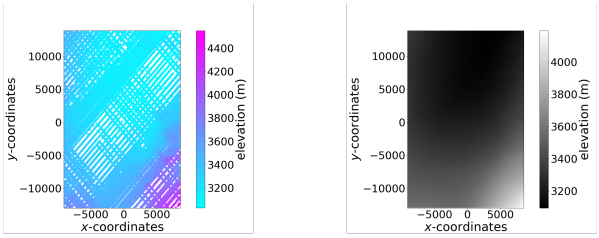
Antoine Lucas
Institut de Physique du Globe de Paris (IPGP), Université Paris Cité, CNRS
Paris, France
lucas@ipgp.fr

ABSTRACT

One of the long-standing question in geoscience is *whether there is a topographical signature of life*. Recent development of space-borne LiDAR has led to massive data depicting planetary topographies, opening up unprecedented opportunities to make progress in answering this question. This is what we set out to do in an ongoing project named *PARKER*. A key step of PARKER is to find topographical features that are potentially relevant to intrinsic differences between Earth and alien worlds. Due to the huge data volume, sequential feature extraction cannot meet our needs in PARKER. Hence, in this work we propose a GPU-accelerated framework for fast feature extraction of planetary LiDAR, which as far as we know is the first GPU-based solution for this task. Faced with multi-scale features and limited GPU memory, we present a novel pseudo-one-pass sweep (POPS) approach, leveraging memory-aware data grouping and incremental data transfer to address these challenges. We also develop a GPU-based solution to aggregate features extracted by POPS. Experiments on real and simulated data show that our algorithms are 2-3 orders of magnitude faster than their sequential counterparts and 1-2 orders of magnitude faster than MPI-based multi-core parallelism, enabling near real-time analytics of datasets with almost a billion points. Based on POPS, we have been able to efficiently evaluate the relevance of topographical features to intrinsic inter-planetary differences. So far, we have assessed the abilities of two feature extractions methods, *PCA* and *STAT*, to capture differences between Earth and Mars. Results show that PCA features on scales of 300-500m can best capture such differences. Thanks to the generic nature of POPS, we will be able to expand our studies to new feature extraction methods and other alien worlds than Mars in the next phase of PARKER.

1 INTRODUCTION

The topography bears the mark of the mechanisms at work on the surface of a planet that shape it. A long-standing question related to topography *whether there a topographical signiture of life* [6]. This is exactly the question we set out to answer in an ongoing project named *PARKER*. PARKER benefits from the growing application of planatary LiDAR in the past two to three decades, which is an advanced remote sensing technology that has lead to datasets portraying the topographies of various planets. (e.g. the GEDI dataset [7] for the Earth and the MOLA dataset [25] for Mars), opening up unprecedented opportunities for inter-planetary comparative studies. Such datasets come in the form of 3D point clouds, in which each point is represented by a tuple (x, y, z), where x and y are longitude and latitude values, while z is the elevation of the point at the aforementioned coordinates [Fig. 1 (a)]. In PARKER, we leverage the profound semantics of planetary LiDAR to find potential topographical signatures of life on Earth, a key step of which is to extract features from LiDAR data of Earth and other planets, so that we can later examine whether any of them point to intrinsic differences between Earth and alien worlds where there are no known life-forms by cross validation (CV) and visualization.



(a) Raw point cloud before interpolation (b) Gridded point cloud after interpolation

Figure 1: An illustration of planetary LiDAR data. (a): raw point cloud from the GEDI [7] dataset, depicting the Socompa Landslide on Earth; (b): gridded point cloud of interpolation with the inverse distance weighting (IDW) [24] algorithm, which is far denser than the raw data.

A major obstacle we face in PARKER is how to efficiently extract features from planetary LiDAR with huge data volume. Covering vast areas, raw planetary LiDAR point clouds can contain hundreds of millions to billions of data points. For example, the GEDI [7] mission is to conduct 10 billion measurements over its nominal two-year mission life, and now extended until January 2023. Worse still, these raw datasets are often irregularly-sampled along the x- and y-axes, which can make it hard to compare data from different geolocations and planets. Hence, it is necessary to transform them into gridded data using some interpolation algorithms [15, 17, 24] (Fig. 1 (b)) which can increase the data volume to several or several dozen times the original volume. Moreover, in PARKER, we would like to achieve near real-time data analytics, where we can obtain the features in a matter of seconds or minutes to quickly evaluate them. CPU-based sequential feature extraction can not meet such criteria for massive data. Therefore, we turn to the Graphics Processing Unit (GPU) [22], a widely-adopted high performance computing hardware whose parallel processing power is ideal for our application.

However, it is not a trivial task to apply GPU acceleration to feature extraction of gridded topographical data. The main challenge is the need for *multi-scale* features, as the same type of features can be exhibited across multiple scales. For example, the distribution of elevation values in a specific geolocation may be constant on square-shaped grids of both 300m × 300m and 500m × 500m. For GPU-based parallelism, an naïve solution to multi-scale feature

extraction is to perform parallel feature extraction on different sub-regions in the grid for each scale, while keeping the processing of multiple scales sequential. However, as will be shown in Section 4.1.1, this requires multiple passes to process all scales, and can often fail to fully unleash the computational power of the GPU.

To solve the multi-scale problem, we propose a novel **psuedo-one-pass sweep (POPS)** framework for feature extraction. With memory-aware data grouping and incremental data transfer, POPS can extract multi-scale features with a nearly one-pass scan (hence the term *psuedo-one-pass*) of the input data, even with the limited GPU memory. Moreover, we note that POPS itself is *not* a feature extraction method. Rather, it is a customizable framework that can be instantiated by feature extraction methods of the user's choosing. As we will show in Section 4.2, even with limited memory, POPS has the potential to adapt to many feature extraction methods.

Apart from feature extraction, in PARKER we need to aggregate the features from multiple sub-regions to obtain location-invariant features. Hence, we also provide a GPU-based method for feature aggregation using various statistics.

The main contributions of this paper are as follows.

- To meet our need for analysis of massive planetary LiDAR data in PARKER, we present POPS, a generic framework for GPU-based feature extraction for massive gridded topographical data. To the best of our knowledge, PARKER is the first project focusing on extracting and comparing topographical features of different planets, and POPS is the first framework for GPU-based planetary LiDAR feature extraction[1]. By processing gridded data in a pseudo-one-pass sweep fashion, POPS promises highly efficient extraction of multi-scale topographical features.
- We showcase the customizability of our POPS framework by demonstrating how to efficiently instantiate it using two specific feature extraction methods, *PCA* and *STAT*, even with highly limited memory resources on the GPU.
- We propose a novel method for aggregating features obtained from POPS on the GPU.
- We conduct extensive experiments on both publicly available real-world data and massive simulated data. Notably, POPS can be over 3 orders of magnitude faster than its sequential counterpart, and 1-2 orders of magnitude faster than MPI-based parallelism with 20 CPU cores.
- We show that in PARKER, POPS can achieve near real-time processing of over 0.7 billion data points, which as far as we know was previously impractical. This has enabled us to efficiently evaluate the relevance of topographical features to intrinsic inter-planetary differences. Assessing the abilities of *PCA* and *STAT* to capture differences between Earth and Mars. we found that *PCA* features on scales of 300-500m can best capture such differences. Moreover, the generic nature of POPS enables us to expand PARKER to new feature extraction methods and other alien worlds than Mars.

The rest of this paper is organizes as follows. Section 2 reviews our related works. Section 3 introduces the preliminaries. Section 4 presents our POPS framework. Section 5 discusses our GPU-based method for feature aggregation. Section 6 provides the experimental results. Finally, Section 7 concludes the paper.

[1]In this paper, we limit our discussion to non-deep features, as opposed to learned features by deep learning as they generally lack the interpretability needed in PARKER.

2 RELATED WORK

For related work, we first look into current works on topographical data that make use of GPU acceleration. A large chunk of these works [3, 18, 19] focus on data interpolation, utilizing GPU-based parallelism to accelerate existing interpolation methods [15, 17, 24]. Apart from these, GPU acceleration has been applied to tasks such as LiDAR point cloud filtering [11], LiDAR data reduction [23] and simulated LiDAR scanning [16]. To the best of our knowledge, there are no existing works dedicated to GPU-accelerated (non-deep) feature extraction for topographical point cloud data.

Next, we review existing works on feature extraction for topographical data. There exist studies that exploit end-to-end deep neural networks for feature learning [8, 10, 13]. Though, these works are beyond the scope of our paper. Surprisingly, to the best of our knowledge, the only work on feature extraction for topographical point cloud data is [2], where Brodu and Lague proposed a PCA-based approach for feature extraction. We will discuss this approach in more detail in Section 4.2.

3 PRELIMINARIES

3.1 Sequential Feature Extraction Framework for Gridded Topographical Data

In PARKER, we rely on cross validation (CV) to evaluate the relevance of a specific feature extraction method. That is, we form a dataset with examples from Earth and Mars, and apply a classifier to predicting which planet each example comes from, using feature extracted by the feature extraction method. The more accurate the predictions, the more relevant the features. Beginning with a handful of large regions of interest (ROIs) from Earth and Mars which is common practice in geoscience, we prepare the data to be fed into the classifier using a *two-level division scheme* shown in Fig. 2.

On the *top* level of the division, we break the ROIs down to sub-regions which will serve as training and testing examples in CV. The reason why we do not use entire ROIs as examples is that there are so few ROIs that they would fail to populate the dataset for meaningful CV. Specifically, we obtain the examples with a square sliding window, which moves upwards and rightwards from the bottom-left corner of the grid [Fig. 2 (a)]. This requires two parameters: the *example scale exScale* which is the edge size of the sliding window (i.e. that of the examples), and the *example step exStep* which is the stride of the window.

On the *bottom* level, we further divide each example into square *patches* of different edge sizes call *feature scales*, on which we will conduct feature extraction. The reason why we do not directly conduct feature extraction on the entire example is that the same topographical semantics can be exhibited across multiple scales, and it is crucial to *simultaneously* consider all of them [2]. We obtain the patches with multiple sliding windows on each example [Fig. 2 (b)(c)]. Again we adopt two parameters: the *feature scales fScales* which are the edge sizes of the windows (i.e. those of the patches), and the *feature steps fSteps* which are the strides of the windows.

With the data divided, we now obtain feature vectors for each example in two steps. First comes *raw feature extraction*, where

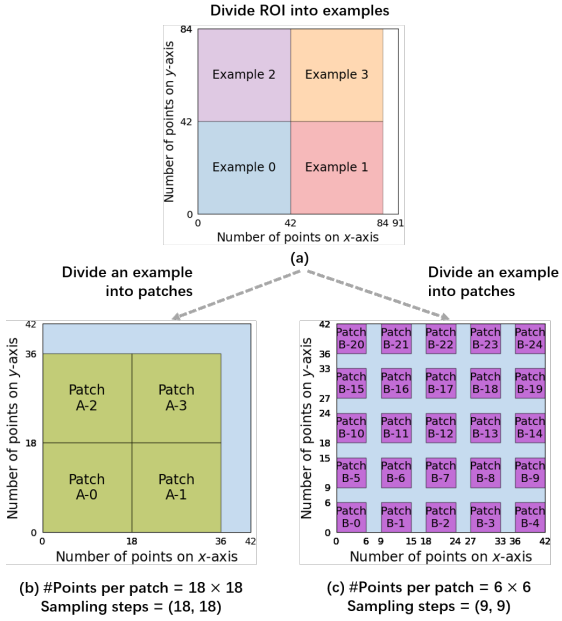


Figure 2: Two-level division of an ROI. (a) Dividing an ROI of size 91×84 into examples of size 42×42 with an example step of 25. For each example, further dividing it into patches of (b) size 18×18 with a feature step of 18, and (c) size 6×6 with a feature step of 9. Note that this is merely a toy example. Actual ROIs and examples are hundreds or thousands of times larger than what is shown here.

we apply a pre-defined feature extraction method called the *core method* to the patches in it. Note again that we are extracting multi-scale features as the patch sizes vary. Also note that the core method is user-defined, hence the customizability of our framework.

Second comes *feature aggregation*, where we aggregate the raw patch-level features to form the final feature vectors of the examples. The reason why we do not directly use the raw features is that they can lead to excessively long feature vectors that are susceptible to curse of dimensionality [12] in CV. To aggregate the features, we first group all patches in each example by their feature scales. For patches of each scale, note that their raw feature vectors share the same dimensionality. On each dimension, we aggregate the feature values of all these patches using 9 statistics: mean, variance, skewness, kurtosis, minimum (min), first quartile (Q1), median, third quartile (Q3), maximum (max). The final feature vector for each example comprises of the 9 statistics for all dimensions of patch-level features under all feature scales.

Before moving on, we discuss parameter settings for the two-level division scheme. The first parameter to set is $fScales$ as this is solely decided by user knowledge about on what scales the core method is most likely to produce high-quality features. With $fscales$ fixed, the choice of $exScale$, $exStep$ and $fStep$ are essentially trade-offs of quantity and quality: The larger $exScale$ is, the more patches an example can hold, and the more robust its aggregated features are likely to be. However, this also means fewer examples, which can limit the performance of the CV classifier. Similarly, the larger $fStep$ and $exStep$ are, the smaller inter-dependency between adjacent patches or examples is, which improves robustness. However, this also means fewer patches or examples. The user needs to carefully consider such trade-offs to set the parameters. Especially, we strongly recommend against setting $exStep$ smaller than $exScale$, as this will cause examples to overlap, breaking the minimal level of independence between examples. However, this does not extend to the setting of $fSteps$ as overlaps of patches in an example does not violate independence of the example.

3.2 CUDA-based GPU Acceleration

In this work, we adopt CUDA-based GPU acceleration to boost the efficiency of feature extraction. CUDA [22] is a widely-used general purpose parallel computing platform and programming model for NVIDIA GPUs, which are especially suitable to process multiple data slices in parallel with a single set of operations, thanks to their single-instruction-multiple-threads (SIMT) architecture, Specifically, a GPU has massive numbers of threads running in parallel on multiple streaming multiprocessors (SMs), and each thread executes the same operations on one data slice. These threads are grouped into **blocks**, which are assigned to SMs to be executed independently from each other.

An important feature of CUDA is its memory hierarchy. Specifically, each GPU thread has its **registers**, all threads in a block can access per-block **shared memory**, and all threads in all blocks can access the same **global memory**. To process data with CUDA, it must first be loaded into global memory, and then be fetched to registers or shared memory by individual threads. The memory size increases in order of per-thread registers, per-block shared memory and global memory, with the size of the latter far greater than the former two. By contrast, accessing registers and shared memory is much faster than accessing global memory, thus global memory accesses should be avoided whenever possible. Rather, when handling data that cannot be fitted into registers, it is advisable to cache it in shared memory Moreover, all data in the shared memory of a block can be accessed by all threads in that block, which provides an effective way of intra-block coordination. Also, CUDA offers the **__syncthreads()** function which serves as an intra-block synchronization barrier where any thread must wait till all other threads in the block have reached the barrier to proceed. See [22] for more information on CUDA programming.

4 GPU-BASED FEATURE EXTRACTION

Our sequential feature extraction workflow can often be inefficient due to the following reason: While the number of ROIs is usually limited, they can often cover large areas. Therefore, for each of them, both the number of examples and the number of patches in each example can be huge. For example, in our studies (see Section 6.1 for the parameter settings we use), the LiDAR data covering area surrounding the Borku drainage system (which is a modest-sized geo-object) on Earth can be divided into 280 examples, while the area of Olympus Mons (the largest mountain in the Solar system) on Mars encompasses 76396 examples. Each of these examples can be further divided into tens or even hundreds of patches. Looping over all of these can be highly time-consuming (in a matter of several days for PARKER, see Section 6.1 for more). In response, we draw on the SIMT architecture of the GPU (Section 3.2) 'to conduct the same feature extraction operations for multiple patches and examples in

parallel. Next, we will mainly present our POPS framework for raw feature extraction, and then discuss parallel feature aggregation.

4.1 Raw Feature Extraction With POPS

4.1.1 Main Challenges. We begin our discussion on GPU-based raw feature extraction by identifying its two main challenges:

Challenge 1: multi-scale features. Given the two-level ROI division scheme shown in Fig. 2, it is intuitive to let each GPU block handle one example, and let each thread in the block handle one patch in the example. The former is advisable as all examples have the same size with the same number of patches, thus the workloads across all blocks are balanced. However, the same cannot be said for allocating one patch to each thread: As the number of threads in each block is limited to 1024 [22] which is often surpassed by the number of patches, it is often required that each thread iteratively process multiple patches. As we adopt multiple feature scales, a naïve way for iterative processing is to go through each feature scale one by one, letting each thread process one or more patches for each scale. However, this comes with the problem of idle threads. Suppose we have a 10000×10000 example, with $fScales$ being 250, 500, 1000 and $fSteps$ the same as the corresponding feature scales for all three of them, thus the number of patches for each scale is 1600, 400, 100. To process the first scale with 1600 patches using 1024 threads, we need two iterations, in the second of which only $1600-1024 = 576$ are busy, while nearly half of the available threads sit idle. For the scales of 500 and 1000, only 400 and 100 threads are busy, resulting in even greater waste of computational resources. It is thus highly desirable to allow different scales to be processed simultaneously to keep as many threads busy for as long as possible.

Challenge 2: limited share memory. Our multi-scale setting dictates that many data points in an example will be accessed multiple times, and directly accessing them from the global memory is highly inefficient. Thus, we leverage shared memory for faster access, yet the example is often too large to be completely fitted into the shared memory. This calls for a method that can both accommodate limited memory space and simultaneous multi-scale processing.

4.1.2 The POPS Framework. We now describe POPS, our GPU-based feature extraction framework that addresses both the challenges mentioned above. As mentioned earlier, each GPU block is assigned to a single example. Raw feature extraction is performed in a **pseudo-one-pass sweep (POPS)** (we will explain the namesake later) of the example. Concretely, we introduce the concept of *superpatches*, which are rectangular areas in the example that can be fitted into the shared memory, with maximum edge sizes of $spScaleX$ and $spScaleY$ on the x- and y-axes. The general idea of POPS is to use several superpatches to sequentially cover all patches in the example, letting threads in the block extract features for the covered patches while the latter reside in shared memory. Note that throughout this process, all superpatches reside in only one chunk of shared memory with a fixed size.

For instance, in Fig. 3 we attempt to extract features for the example that is divided into patches as shown in Fig. 2 *bottom)*. With $spScaleX$ and $spScaleY$ being both 21, the first superpatch (Superpatch 0) is initiated at the bottom-left corner of the example. This can be done by letting each thread in the block transfer one or several data points from global memory to the chunk of shared memory allocated to the superpatch. Starting here, we iteratively conduct the following two steps.

Step 1: apply the core method to all patches of all scales in the current superpatch. Here we let each thread extract features for one or several patches. For example, in Fig. 3, Superpatch 0 covers Patches A-0, B-0, B-1, B-5, B-6 in Fig. 2 (b)(c). Suppose we have 3 threads, then we let each thread execute the core method on A-0, B-0, B-1, and also let the first 2 threads handle the B-5, B-6. Note that we allow different threads to handle patches of different feature scales simultaneously, thus avoiding the aforementioned waste of resources issue with sequential processing of each feature scale.

Step 2: move to the next superpatch in one of two ways: One is the *go-right* move, namely move to the next superpatch in the same row along the x-axis. The x-coordinate of the leftmost points in the next superpatch is that of the first patch in this row that the current superpatch fails to cover. For example, in Fig. 3, the first patches that Superpatch 0 fails to cover on the x-axis are A-1 and B-2 which share the same leftmost x-coordinate of 18, which is that of Superpatch 1. Go-right moves continue till the last possible superpatch in the current row, namely Superpatch 2.

For the go-right move, we can use a simple *incremental data transfer* strategy to negate the need for repeated global memory access. Specifically, we store the data points in each superpatch in a column-first (i.e. the y-coordinates of contiguous points change faster) manner in the shared memory. This means that when we perform a go-right move, if the next and the current superpatches have any overlap (i.e. the first few columns of data points in the new superpatch also form the last few columns in the previous one), these data points already reside in a contiguous sub-chunk of the chunk of shared memory allocated to the superpatches. Hence, we do not need to re-fetch these points from global memory. Rather, we can simply move the starting index of the new superpatch in the shared memory to where these data points reside. For example, Fig. 4 shows the incremental data transfer for move from Superpatch 0 to 1. In Fig. 3, the first 3 columns of Superpatch 1 are also the last 3 columns of Superpatch 0, and are already in a sub-chunk of shared memory with the starting index of $18 \times 21 = 378$. Thus, we simply move the starting position of Superpatch 0 to 378 without reloading these 3 columns. For the rest of the new superpatch, we append these data points to the end of the existing ones. If we hit the end of the chunk of shared memory for superpatches, we simply re-use the indices from the beginning of the chunk. For example, for Superpatch 1, the columns other than the first 3 ones are stored in the positions with indices 0 to $18 \times 21 - 1 = 377$.

The second way of moving is the *go-up* move. When a go-right move hits the end of a row, we re-initiate the superpatch at the leftmost position of the next row. The y-coordinate of the points at the bottom of the current row is the minimum y-coordinate among the points in patches that have not been covered by previous rows. For example, in Fig. 3, when moving from Superpatch 2 to 3, this minimum y-coordinate is 18, which is that of the points at the bottom of Patches A-2, A-3, B-10, B-11, B-12, B-13, and B-14. Thus, Superpatch 3 starts at 18 on the y-axis. After the go-up move, we can again perform the go-right move. The iterative process of "go-right, then go-up" stops when all patches have been processed.

Looking back on the entire pseudo-one-pass sweep, we reach a point where we can explain its namesake. By *one-pass*, we mean that

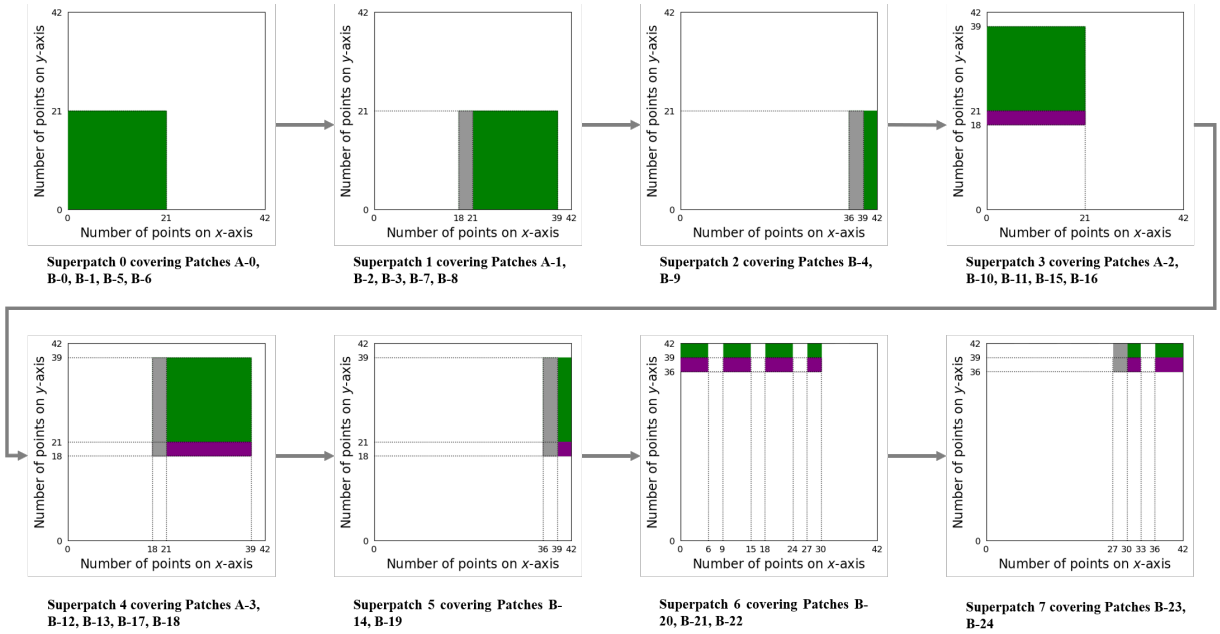


Figure 3: An illustration of our pseudo-one-pass sweep (POPS) framework. For the example that is divided into patches in Fig. 2 (b)(c), we use superpatches with a maximum size of 21×21 to moving over all patches in it, fetching data points from global memory to shared memory. Within each superpatch, the green regions indicate data points that have *not* been covered by any previous superpatches. The grey regions indicate data points that have been covered by the last superpatch in the same row, but have *not* by any superpatches in a previous row; by incremental data transfer (Fig. 4), we can avoid repeated global memory access to re-fetch these data points to the current superpatch. The purple regions indicate data points that have been covered by a superpatch in a previous row, which require repeated global memory access to be fetched for the current superpatch.

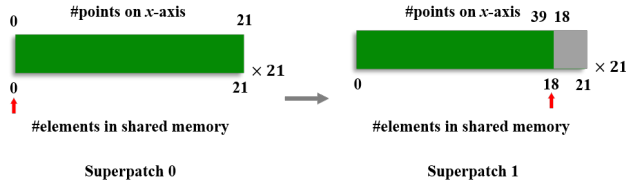


Figure 4: An illustration of shared memory usage for Superpatches 0-1 under incremental data transfer. Green indicates new data points that require global memory access. Grey indicates data that are already in shared memory. Red arrow indicates where the first point in the superpatch is stored.

unlike the aforementioned naïve approach where we scan the global memory multiple times for all feature scales, we can now *logically* perform only one scan with the superpatches for multi-scale feature extraction. By *pseudo* we mean while it is *logically* possible to scan only once, in practice we need repeated global memory access for overlapping data points in superpatches from different rows (the purple regions in Fig. 3). However, the number of such points are usually very limited compared to the total number of points in the example. Also, our incremental data transfer strategy ensures that *no repeated global memory access is required for a single row*, which can greatly reduce the total number of repeated accesses. Also, from a methodological perspective, faced with multi-scale patches, rather than simply group them by their scale, POPS achieves a memory-aware grouping with superpatches, thus achieving a higher degree of parallelism under limited shared memory resources.

Before we move on, we discuss the setting of the superpatch size parameters $spScaleX$ and $spScaleY$. The idea is to make the superpatch as large as possible given a fixed-sized chuck of shared memory to hold it. This can be done by making $spScaleX$ and $spScaleY$ as large as possible while keeping their difference as small as possible. That is, if the chunk of memory can hold up to N datas points, we let $spScaleX$ and $spScaleY$ to be roughly $\sqrt{N}$. Also, note that a superpatch must be larger than the largest of all patches sizes, which poses a limitation on the setting of $fScales$ as under the largest scale, the number of points in the patch must be smaller than $spScaleX \times spScaleY$. However, we find this limitation to have little impact in PARKER (Section 6.1) where $fScales$ are set by expert knowledge.

4.2 Memory-efficient Core Method Implementation

As mentioned in Section 3.1, our feature extraction framework allows for user-defined core method. On the CPU where memory resources are usually abundant, there are few limitations to the choice of core method. We now show that this also holds on the GPU where memory space is relatively scarce. Recall that we let each thread execute the core method for a single patch at a time. The patch itself, which holds the raw gridded data, already resides in shared memory. Thus, the only remaining concern is assigning the *overhead workspace memory* needed by the core method to process the raw data. Given the limited size of per-thread registers, we need

to relegate the workspace to shared or global memory. Modern GPUs have tens of GBs of global memory, which allows the vast majority of (if not all) core methods to be run on the GPU with their workspace in global memory. However, as global memory is slow to access, we opt to assign the workspace within the registers (and a small chunck of the shared memory as needed). This requires the workspace to be of $O(1)$ size, regardless of the number of points in raw data. As it turns out, in many cases, this is achievable.

To illustrate, we look into two core methods we use in PARKER, and show how they can be implemented with $O(1)$ workspace with low time complexity.

Core method 1: *PCA.* In [2], Brodu and Lague proposed to use the explained variance ratios of the first two PCA components of a given patch as its features. This entails 3 steps: 1) Obtain the covariance matrix $\mathbf{C}$ of the patch. Suppose the patch contains n raw points, each being a 3D coordinate, then the patch can be represented as an $n \times 3$ matrix $\mathbf{M} = [\mathbf{m}_0^{\mathrm{T}}, \mathbf{m}_1^{\mathrm{T}}, \mathbf{m}_2^{\mathrm{T}}]$ where $\mathbf{m}_i^{\mathrm{T}} = [m_{i,1}, m_{i,2}, \ldots, m_{i,n}]$ $(i = 1, 2, 3)$ are columns of $\mathbf{M}$, and $\mathbf{C}$ is a 3×3 matrix where the element at position (i, j) is

$$c_{i,j} = \frac{\sum_{k=1}^{n} m_{i,k} m_{j,k}}{n-1} \tag{1}$$

Note that $c_{i,j}$ can be obtained in $O(n)$ time with $O(1)$ space as we only need to keep one $m_{i,k}$ and one $m_{j,k}$ in the workspace at a time. As there are $3 \times 3 = O(1)$ elements in $\mathbf{C}$, it takes $O(n)$ time and $O(1)$ space to obtain $\mathbf{C}$. 2) Obtain the eigenvalues λ_1, λ_2 and λ_3 of $\mathbf{C}$, which takes $O(1)$ time and space as the size of $\mathbf{C}$ is constant. 3) Obtain the explained variance ratios which are $\frac{\lambda_1}{\lambda_1+\lambda_2+\lambda_3}$ and $\frac{\lambda_2}{\lambda_1+\lambda_2+\lambda_3}$, which can also be obtained in $O(1)$ time and space. Thus the entire process takes $O(1)$ workspace and $O(n)$ time.

Core method 2: *STAT.* The same statistics used for feature aggregation (Section 3.1) can also be used as raw features when applied to the elevation (namely z) values in a patch (rather than to the raw features of the patch as in feature aggregation). Suppose the set of elevation values are $\{z_i \mid 1 \le i \le n\}$. Among the aforementioned statistics, mean, variance, skewness and kurtosis (denoted as μ, σ^2, $\tilde{\mu}_3$ and $\tilde{\mu}_4$) can be calculated as follows:

$$\mu = \frac{\sum_i z_i}{n} \tag{2}$$

$$\sigma^2 = \frac{\sum_i z_i^2}{n} - \mu^2 \tag{3}$$

$$\tilde{\mu}_3 = \frac{\sum_i z_i^3 - 3\mu \sum_i z_i^2 + 3\mu^2 \sum_i z_i - n\mu^3}{n\sigma^3} \tag{4}$$

$$\tilde{\mu}_4 = \frac{\sum_i z_i^4 - 4\mu \sum_i z_i^2 + 6\mu^2 \sum_i z_i^2 - 4\mu^3 \sum_i z_i + n\mu^4}{n\sigma^4} - 3 \tag{5}$$

We can obtain the sums $\sum_i z_i$, $\sum_i z_i^2$, $\sum_i z_i^3$, $\sum_i z_i^4$ in these equations in $O(n)$ time and $O(1)$ workspace, as we only need to keep the sum so far and the next z_i value in the workspace at a time. With these obtained, the 4 statistics can be trivially computed in O(1) time and space. Thus, it takes $O(1)$ workspace and $O(n)$ time in total.

For min and max, we can obtain them in O(1) workspace and O(n) time with a one-pass scan of the patch to fetch the minimum and maximum z values. For Q1, median and Q3, we can obtain them using some selection algorithm which usually comes with a space-time trade-off. In our implementation, we use the *basic algorithm* proposed in [20], which uses $O(1)$ workspace and $O(nlgn)$ time. This is fast enough in most cases where the patch sizes are limited. In the rare cases where this is too slow, Munro and Raman [20] provided faster methods with slightly higher workspace requirement. We omit discussion of these methods for brevity.

5 GPU-BASED FEATURE AGGREGATION

We now discuss GPU-based feature aggregation, which entails calculation of the statistics mentioned in Section 3.1 over the raw features. For mean, variance, skewness and kurtosis, similar to what we did for *STAT* in Section 4.2, we can first obtain the 4 sums and then trivially calculate these statistics. The sums can be obtained by CUDA-based parallel reduction [9]. For min, Q1, mean, Q3 and max, we sort each dimension of the patch-level features and obtain them trivially, using the CuPy [21] library for GPU-based sorting. After pre-computing the sums and the sorted raw features, we let each GPU thread handle the (trivial) calculation of a single aggregated statistic, thus completing feature aggregation.

6 EXPERIMENTAL EVALUATION

For experiments, we first showcase the utility of the proposed algorithms in our PARKER project. Then, we zoom in on our POPS framework and see how different grid configurations and parameter settings can affect its efficiency. Unless otherwise stated, all CPU-based experiments were run on a Ubuntu 18.04 server with 1 AMD EPYC 7402P 24-core CPU @2.8GHz and 125GB memory, while all GPU-based experiments were run on a *laptop computer* with 1 GeForce GTX 1660 Ti GPU and Windows 10 OS. The idea is to show that even on a laptop computer, our GPU-based algorithms can still be much faster than CPU-based ones on a dedicated server. All CPU-based code is written in Python 3, while all GPU-based code is written in CUDA-C [22] and linked to the Python code using PyCUDA [14]. The number of threads per GPU block is 256 throughout the experiments. All results are averaged over 10 runs.

6.1 Utility in PARKER

As mentioned in Section 3.1, to find potential topographical signatures of terrestrial life, we compare the topographical features extracted from Earth with other planets. Currently, we are interested in comparing *PCA* and *STAT* on their abilities to find features that can distinguish between the Earth and Mars. To do so, we select 20 ROIs (11 from the Earth, 9 from Mars) from the GEDI [7] and MOLA [26] repositories which contain point clouds of both planets respectively. Using 5 interpolation methods to preprocess the raw data, we end up with a dataset of 100 data grids with a total number of 702,223,010 data points. The resolution (i.e. distance between adjacent grid points on x- and y-axes) of the grids is 60 m.

Under the guidance of a geoscientist, we pinpoint 3 groups of $fScales$ on which characteristic features likely exist into the data: the *small* scales of (300m, 400m, 500m), the *medium* scales of (700m, 800m, 900m), and the *large* scales of (1100m, 1200m, 1300m). Then, following the trade-off of example (feature) quantity and quality mentioned in Section 3.1, we set $exScale$ and $exStep$ to be both 2000m and $fSteps$ to be half of $fScales$. We run both *PCA* and *STAT* using this setting with the following frameworks: 1) *CPU* which is the sequential framework; 2) *MPI* which is used for parallel processing with 20 cores, each core handling some of the examples in the current ROI. 3) *POPS*, running under two hardware settings: the laptop-based GTX 1660 Ti GPU for efficiency evaluation only,

Core method	Raw feature extraction time (s)				Feature aggregation time (s)		
	CPU	MPI	POPS-NMS	POPS	CPU	MPI	GPU
PCA	$6.2 \times 10^4 \pm 1.1 \times 10^3$	$3.2 \times 10^3 \pm 2.7 \times 10^2$	$4.9 \times 10^1 \pm 6.7 \times 10^{-2}$	$3.8 \times 10^1 \pm 1.2 \times 10^{-1}$	$4.6 \times 10^3 \pm 3.3 \times 10^2$	$2.7 \times 10^2 \pm 1.9 \times 10^1$	$6.7 \times 10^1 \pm 2.1 \times 10^{-1}$
STAT	$1.4 \times 10^5 \pm 8.0 \times 10^2$	$7.1 \times 10^3 \pm 6.8 \times 10^2$	$3.0 \times 10^3 \pm 1.6 \times 10^0$	$1.6 \times 10^3 \pm 4.4 \times 10^0$	$1.8 \times 10^4 \pm 1.3 \times 10^2$	$1.1 \times 10^3 \pm 8.9 \times 10^1$	$1.6 \times 10^2 \pm 1.6 \times 10^1$
Total	2.0×10^5	1.0×10^4	3.1×10^3	1.6×10^3	2.3×10^4	1.4×10^3	2.2×10^2

Table 1: Running time on PARKER data (mean ± std)

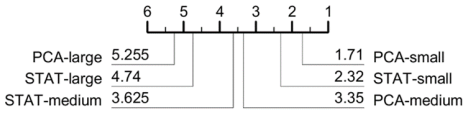


Figure 5: Critical difference diagram of cross validation results in PARKER.

and a server-based Quadro RTX 6000 GPU which we actually used in PARKER. 4) *POPS-NMS* which is POPS with simultaneous processing of multi-scale features disabled by the _syncthreads() function, also run on the two different GPUs. For feature aggregation, we consider CPU-based sequential aggregation, MPI-based aggregation with 24 cores, and GPU-based aggregation on both hardwares.

The running times on the 100 grids are shown in Table 1. For raw feature extraction, MPI is about 20x faster than CPU as 20 cores were used, yet it is still no match for our POPS, which is over 1600x and 80x as fast as CPU and MPI for *PCA*, and over 85x and 4x as fast for *STAT*. The smaller gain on *STAT* is likely because we used an $O(nlgn)$ algorithm [20] to calculate Q1, median and Q3, rather than an $O(n)$ one that is relatively easy to implement on CPU. Still, our acceleration is substantial, despite running on merely a laptop computer. Also, POPS is almost 2x as fast as POPS-NMS, which can make a great difference when the absolute running time is long, such as the case with *STAT*. Finally, for feature aggregation, our method is over 100x and 6x as fast as CPU and MPI.

In Table 1, we deliberatedly ran our GPU-based algorithms on a laptop to highlight their efficiency. In PARKER, we use a server with a Quadro RTX 6000 GPU instead, leading to about 592s for raw feature extraction and 6.52s for aggregation; in particular, for *PCA*, the times are 1.31s and 1.30s which are near instant. This makes near real-time analytics of large topographical data possible, which to the best of our knowledge has never been achieved before. By *near real-time* we mean providing results within a matter of seconds or minutes, as such delays are negligible when compared to data collection time and cause minimal inconvenience to researchers. Such near real-time processing enables us to efficiently compare the abilities of *PCA* and *STAT* to distinguish between Earth and Mars under the 3 aforementioned $fScales$ settings. Specifically, we perform the comparison using cross validation (CV) with a range of settings for train-test split, data balancing and classifier, using the Cohen's kappa coefficient [4] as the evaluation metric. Following the practice of Bagnall et al. [1], we conduct Wilcoxon signed rank tests to validate the significance of the CV results, which are shown in Fig. 5 in a critical difference diagram [5]. Here, methods to the right of the diagram perform better, and a bold black bar will be used to group two or more methods together if they do not have statistically significant differences (this is not present in Fig. 5). As is shown, *PCA* under the *small* $fScales$ (300m, 400m, 500m) outperform all other methods, and *PCA* is generally superior to *STAT*. This is also validated by visualization studies. For example, Fig. 6 visualizes the aggregated features of the Socompa Landslide on Earth [7] using the t-SNE [27] method, with all three groups of $fScales$ concatenated into one feature vector for each example. As is shown, the data distribution for *PCA* is denser and more regular.

The potential of POPS does not end here. Due to its generic nature, it allows us to efficiently test new feature extraction methods and compare Earth with other align worlds than Mars in the next phase of PARKER , which may finally answer the question: is there a topographical signature of life on earth?

6.2 Experiments on Simulated Data

As is shown in Table 1, the bulk of the total feature extraction time for both CPU- and GPU-based solutions is spent on raw feature extraction, thus we further evaluate our POPS raw feature extraction framework on simulated data grids. For the first simulation, as the choice of feature scales $featScales$ is of great geoscientific importance in our PARKER project, we examine how POPS handles different settings of $featScales$ with varying number of examples and number of points in each example. Specifically, we generate a grid with $30000 \times 30000 = 9 \times 10^8$ points. Fixing $exStep$ to be the same as $exScale$, we set different values for $exScale$ and thus different numbers of examples in the grid. For $featScales$, we consider three options: *small* where the patches have 5-10 points along each of the two edges, *medium* with 15-20 points, and *large* with 25-30 points. Note that these numbers are NOT the number of points in the patch (which is the square of the former), and are NOT the same as the *small*, *medium* and *large* settings in the previous section, although the former do mimic the latter to simulate real-world cases. With $featSteps$ fixed to half of $featScales$ which is the same as our real-world configuration, we run POPS under these 3 settings and record the running times.

The results are shown in Fig. 7. First of all, similar to what was observed on real data, *STAT* is much slower than *PCA* due to its higher time complexity. As with the $featScales$ settings, both *small* and *medium* leads to relatively short running time while *large* is significantly slower. This reveals a weakness of our POPS methods: It only allows one thread per patch, no matter how large the patch is, thus it is relatively inefficient at processing large feature scales. In particular, despite its optimizations for multi-scale scenarios, this yields limited gains when *all* the scales are large. That being said, POPS is still reasonably efficient under the *large* setting. Another interesting phenomenon is the zig-zag pattern of the running time curve for *large*. This is likely the combined effect of two competing factors: As the number of points in each example increases, so does the number of patches per example and thus the per-example processing time. On the other hand, the number of examples drops as they grow larger, which decreases the overall running time.

Next we consider the effect of the grid size on raw feature extraction time. Similar to what we did on real data, here we consider the following three methods: CPU, POPS-NMS, POPS. Fixing the number of points in each example to 300 and letting $exStep = exScale$, we let $featScales$ to be the union of the *small*, *medium* and *large*

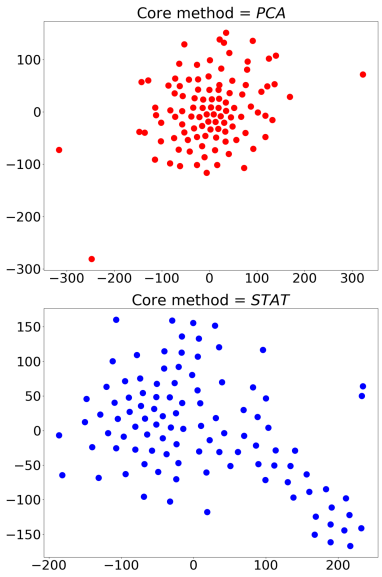


Figure 6: Visualization of aggregated features of the Socompa Landslide on Earth.

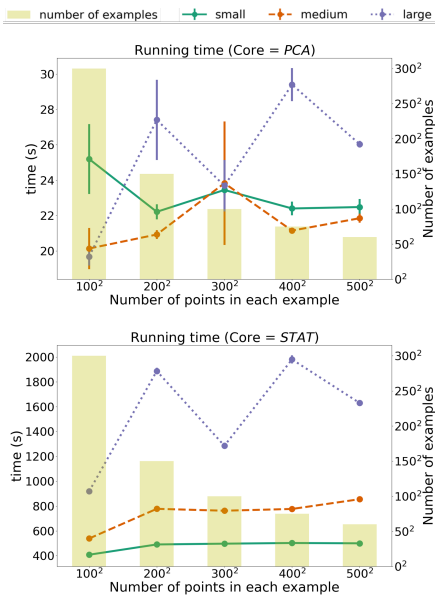


Figure 7: Running time of POPS on a simulated grid with 30000×30000 **points.**

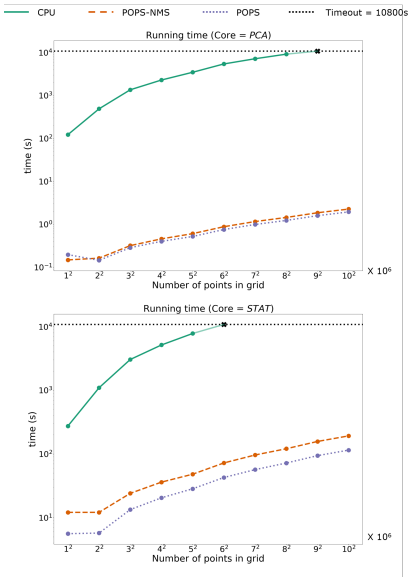


Figure 8: Raw feature extraction time on simulated grids with different sizes. Note that the time axis is in log scale.

settings in the previous experiment and keep $featSteps$ as being half of $featScales$. Then we set different grid sizes and record the running time. Note that we set a timeout threshold of 3h (10800s).

The results are shown in Fig. 8. For the CPU-based solution, it is much slower than the GPU-based frameworks, running out of time for *PCA* and *STAT* before reaching a grid size of $9^2 \times 10^6 = 8,100,000$ points. Also, in terms of the trend of the curves for GPU-based solutions and the CPU-based one, for *PCA* the trend are similar. However, for *STAT*, the difference between the two are relatively small at the beginning, yet gradually grows as the grid gets larger. This is likely because when the grid is small, the power of parallelism relatively weaker at offseting the disadvantage we have in terms of per-patch time complexity. However, as the grid becomes larger, the power of parallelism gradually surpasses the effect of higher time complexity, thus leading to greater advantage on our side. Finally, in the vast majority of cases (especially when the grid is larger), POPS is able to beat POPS-NMS thanks to the optimizations for multi-scale features.

7 CONCLUSIONS

In this paper, in the context of the ongoing PARKER project where we search for topographical signatures of life on Earth, we proposed POPS, an original and generic framework for GPU-based feature extraction for 3D data and applied it to planetary LiDAR data on both the Earth and Mars. POPS is optimized for multi-scale feature extraction with limited GPU memory resources, and scales to massive LiDAR datasets. We showcased its customizability by demonstrating how to instantiate it efficiently with two core methods *PCA* and *STAT*. We also proposed a method for GPU-based feature aggregation. Extensive experiments have demonstrated the efficiency of our algorithms, which allow for the first time near real-time analytics of massive planetary LiDAR data. With POPS, we were able to compare the abilities of *PCA* and *STAT* to distinguish between Earth and Mars, where we found *PCA* the superior method. POPS will also help us test new feature extraction methods and extend to new alien worlds in the next phase of PARKER. Also, it is worth noting that our algorithms are also applicable to non-planetary LiDAR data. Essentially, they are suitable for any gridded data in the 3D space that supports the two-level division scheme introduced in Section 3.1. For example, it can also be used to extract features from airborne Radar data of the Earth surface, and analyze data from a neutrino telescopes which has an underwater array of gridded sensors in the 3D space.

REFERENCES

[1] Anthony Bagnall, Jason Lines, Aaron Bostrom, James Large, and Eamonn Keogh. 2017. The great time series classification bake off: a review and experimental evaluation of recent algorithmic advances. *Data mining and knowledge discovery* 31, 3 (2017), 606–660.

[2] Nicolas Brodu and Dimitri Lague. 2012. 3D terrestrial lidar data classification of complex natural scenes using a multi-scale dimensionality criterion: Applications in geomorphology. *ISPRS Journal of Photogrammetry and Remote Sensing* 68 (2012), 121–134.

[3] Tangpei Cheng. 2013. Accelerating universal Kriging interpolation algorithm using CUDA-enabled GPU. *Computers & geosciences* 54 (2013), 178–183.

[4] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[5] Janez Demšar. 2006. Statistical comparisons of classifiers over multiple data sets. *The Journal of Machine Learning Research* 7 (2006), 1–30.

[6] W. Dietrich and J. Perron. 2006. The search for a topographic signature of life. *Nature* 439 (2006), 411–418. https://doi.org/10.1038/nature04452

[7] Ralph Dubayah, James Bryan Blair, Scott Goetz, Lola Fatoyinbo, Matthew Hansen, Sean Healey, Michelle Hofton, George Hurtt, James Kellner, Scott Luthcke, et al. 2020. The Global Ecosystem Dynamics Investigation: High-resolution laser ranging of the Earth's forests and topography. *Science of remote sensing* 1 (2020), 100002.

[8] Haiyan Guan, Yongtao Yu, Zheng Ji, Jonathan Li, and Qi Zhang. 2015. Deep learning-based tree classification using mobile LiDAR data. *Remote Sensing Letters* 6, 11 (2015), 864–873.

[9] Mark Harris et al. 2007. Optimizing parallel reduction in CUDA. *Nvidia developer technology* 2, 4 (2007), 70.

[10] Danfeng Hong, Lianru Gao, Renlong Hang, Bing Zhang, and Jocelyn Chanussot. 2020. Deep encoder-decoder networks for classification of hyperspectral and LiDAR data. *IEEE Geoscience and Remote Sensing Letters* (2020).

[11] Xiangyun Hu, Xiaokai Li, and Yongjun Zhang. 2012. Fast filtering of LiDAR point cloud in urban areas based on scan line segmentation and GPU acceleration. *IEEE Geoscience and Remote Sensing Letters* 10, 2 (2012), 308–312.

[12] Gordon Hughes. 1968. On the mean accuracy of statistical pattern recognizers. *IEEE transactions on information theory* 14, 1 (1968), 55–63.

[13] Shichao Jin, Yanjun Su, Shang Gao, Fangfang Wu, Qin Ma, Kexin Xu, Tianyu Hu, Jin Liu, Shuxin Pang, Hongcan Guan, et al. 2019. Separating the structural components of maize for field phenotyping using terrestrial LiDAR data and deep convolutional neural networks. *IEEE Transactions on Geoscience and Remote Sensing* 58, 4 (2019), 2644–2658.

[14] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. 2012. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Comput.* 38, 3 (2012), 157–174.

[15] Daniel G Krige. 1951. A statistical approach to some basic mine valuation problems on the Witwatersrand. *Journal of the Southern African Institute of Mining and Metallurgy* 52, 6 (1951), 119–139.

[16] Alfonso López, Carlos J Ogayar, Juan M Jurado, and Francisco R Feito. 2022. A GPU-accelerated framework for simulating LiDAR scanning. *IEEE Transactions on Geoscience and Remote Sensing* (2022).

[17] George Y Lu and David W Wong. 2008. An adaptive inverse-distance weighting spatial interpolation technique. *Computers & geosciences* 34, 9 (2008), 1044–1055.

[18] Gang Mei. 2014. Evaluating the power of GPU acceleration for IDW interpolation algorithm. *The Scientific World Journal* 2014 (2014).

[19] Gang Mei, Nengxiong Xu, and Liangliang Xu. 2016. Improving GPU-accelerated adaptive IDW interpolation algorithm using fast kNN search. *SpringerPlus* 5, 1 (2016), 1–22.

[20] J Ian Munro and Venkatesh Raman. 1996. Selection from read-only memory and sorting with minimum data movement. *Theoretical Computer Science* 165, 2 (1996), 311–323.

[21] ROYUD Nishino and Shohei Hido Crissman Loomis. 2017. Cupy: A numpy-compatible library for nvidia gpu calculations. *31st confernce on neural information processing systems* 151 (2017).

[22] NVIDIA. 2021. CUDA C++ Programming Guide. (2021).

[23] Dossay Oryspayev, Ramanathan Sugumaran, John DeGroote, and Paul Gray. 2012. LiDAR data reduction using vertex decimation and processing with GPGPU and multicore CPU technology. *Computers & Geosciences* 43 (2012), 118–125.

[24] Donald Shepard. 1968. A two-dimensional interpolation function for irregularly-spaced data. In *Proceedings of the 1968 23rd ACM National Conference*. 517–524.

[25] David E Smith, Maria T Zuber, Herbert V Frey, James B Garvin, James W Head, Duane O Muhleman, Gordon H Pettengill, Roger J Phillips, Sean C Solomon, H Jay Zwally, et al. 2001. Mars Orbiter Laser Altimeter: Experiment summary after the first year of global mapping of Mars. *Journal of Geophysical Research: Planets* 106, E10 (2001), 23689–23722.

[26] David E Smith, Maria T Zuber, Sean C Solomon, Roger J Phillips, James W Head, James B Garvin, W Bruce Banerdt, Duane O Muhleman, Gordon H Pettengill, Gregory A Neumann, et al. 1999. The global topography of Mars and implications for surface evolution. *Science* 284, 5419 (1999), 1495–1503.

[27] Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research* 9, 11 (2008).