International Journal of Geo-Information

MDPI

Article

A Hybrid Parallel Spatial Interpolation Algorithm for Massive LiDAR Point Clouds on Heterogeneous CPU-GPU Systems

Hongyan Wang, Xuefeng Guan * and Huayi Wu

The State Key Laboratory of Information Engineering in Surveying, Mapping and Remote Sensing, Wuhan University, 129 Luoyu Road, Wuhan 430079, China; wanghongyan@whu.edu.cn (H.W.); wuhuayi@whu.edu.cn (H.W.)

* Correspondence: guanxuefeng@whu.edu.cn; Tel.: +86-137-2021-1880

Received: 30 September 2017; Accepted: 14 November 2017; Published: 16 November 2017

Abstract: Nowadays, heterogeneous CPU-GPU systems have become ubiquitous, but current parallel spatial interpolation (SI) algorithms exploit only one type of processing unit, and thus result in a waste of parallel resources. To address this problem, a hybrid parallel SI algorithm based on a thin plate spline is proposed to integrate both the CPU and GPU to further accelerate the processing of massive LiDAR point clouds. A simple yet powerful parallel framework is designed to enable simultaneous CPU-GPU interpolation, and a fast online training method is then presented to estimate the optimal decomposition granularity so that both types of processing units can run at maximum speed. Based on the optimal granularity, massive point clouds are continuously partitioned into a collection of discrete blocks in a data processing flow. A heterogeneous dynamic scheduler based on the greedy policy is also proposed to achieve better workload balancing. Experimental results demonstrate that the computing power of the CPU and GPU is fully utilized under conditions of optimal granularity, and the hybrid parallel SI algorithm achieves a significant performance boost when compared with the CPU-only and GPU-only algorithms. For example, the hybrid algorithm achieved a speedup of 20.2 on one of the experimental point clouds, while the corresponding speedups of using a CPU or a GPU alone were 8.7 and 12.6, respectively. The interpolation time was reduced by about 12% when using the proposed scheduler, in comparison with other common scheduling strategies.

Keywords: CPU-GPU; hybrid parallel algorithm; spatial interpolation; thin plate spline; LiDAR point clouds

1. Introduction

Spatial interpolation (SI) is a well-studied spatial analysis functionality in GIS for deriving a smoothed surface from a limited but usually large number of scattered sample points. SI analysis can help researchers understand implicit spatial trends of geographical phenomena, such as elevation, rainfall, or chemical concentrations. A variety of SI methods have been developed [1], including Inverse Distance Weighting (IDW), Natural Neighbor, and Spline and Kriging. These interpolation methods, however, are usually computationally expensive and time-consuming. Furthermore, given the rapid and ongoing development of spatial data acquisition technologies, an explosive increase in data volume has dramatically aggravated the performance issues confronting SI algorithms. For example, the point density of airborne LiDAR data can reach up to 100 pts/m^2, and thus easily generates billions of points [2]. Faced with these massive point clouds, traditional sequential SI methods cannot perform efficiently.

To address these challenges, numerous acceleration methods have been put forward to take advantage of available parallel resources. In the last decades, most parallel SI algorithms were

implemented in super computing systems, e.g., MasPar [3], Cray T3D [4], CM5 [5], and parallel clusters [6]. While being highly efficient, these methods were implemented on high-end hardware, and thus were unaffordable and inaccessible to normal users. With the emergence and development of multicore CPUs, many researchers began to accelerate SI algorithms on these cost-effective and accessible platforms. For example, Cheng et al. parallelized universal Kriging interpolation based on OpenMP [7]; Guan et al. [8] accelerated the IDW algorithm by leveraging the power of multicores. In recent years, the rapid rise of general-purpose GPUs (GPGPU) has attracted considerable attention for their superior computing power, high memory bandwidth, and low power consumption. As a result, more and more SI algorithms, such as ordinary Kriging [9], universal Kriging [10], and Natural Neighbor interpolation [11] have been further accelerated on many-core GPU platforms.

Modern commodity computers including laptops, desktops, and high performance workstations, are usually equipped with both multicore CPUs and many-core GPUs. These heterogeneous computing systems are ubiquitous, but current CPU-only and GPU-only algorithms exploit just one type of parallel processor, thus resulting in a waste of computing resources. To the best of our knowledge, few studies have explored the acceleration of SI algorithms simultaneously leveraging both types of Processing Units (PUs).

In this paper, we address the gap and propose a hybrid parallel algorithm that parallelizes the Thin Plate Spline (TPS) algorithm to speed up spatial interpolation from massive LiDAR point clouds. On heterogeneous platforms, the hardware architecture, programming models, and computing power of multicore CPUs and many-core GPUs are dramatically different. The integration challenges therefore include unavailable uniform programming models, inefficient CPU-GPU collaboration, and task scheduling. Our research focuses on addressing the last two challenges. A parallel interpolation framework based on a hybrid-programming model was designed to leverage the computing power of both CPU and GPU. Based on this framework, a fast online training method for computing capability estimation is proposed for rapidly finding the optimal task decomposition granularity to keep each PU running at maximum speed. Workload balance is realized with a Heterogeneous Dynamic Scheduler based on the Greedy policy (HDSG), which supports data subdivision. This HDSG policy also helps to improve GPU utilization by task priority declaration. Experimental results demonstrate that both PUs are fully exploited under the optimal granularity, and the hybrid parallel TPS algorithm obtains significant performance improvement. For example, the highest speedup achieved on one of the experimental point clouds was about 20.2. In contrast, the corresponding speedups of the CPU-only and GPU-only algorithms were about 8.7 and 12.6, respectively. Using our HDSG policy, the interpolation time was reduced by about 12% in comparison with other common scheduling strategies (i.e., greedy policy, work stealing).

The rest of this paper is organized as follows: Section 2 reviews the background of the CPU/GPU hardware architecture, programming models, and heterogeneous computing. Section 3 introduces TPS interpolation algorithm and the transformation to local TPS. Section 4 explains the details of the proposed hybrid parallel interpolation framework, including fast online training, streaming spatial decomposition, and the HDSG scheduling strategy. Section 5 presents a performance evaluation and discussion. Finally, Section 6 draws conclusions and outlines directions for future research.

2. Background

2.1. CPU/GPU Architecture and Programming

In a heterogeneous CPU-GPU system, the architecture of the CPU and GPU are quite different. As shown in Figure 1 (adapted from [12]), a CPU core uses a large fraction of the die area for the cache and control logic, and leaves only a small part for integer and floating-point execution (i.e., Arithmetic Logic Units, ALU). Large-sized caches enable low-latency access to cached data sets, while the control unit is dedicated to out-of-order and speculative execution. Thus, a CPU is optimized for single-threaded performance and is suitable for latency-critical applications. As CPUs move toward

multicore architectures, modern CPUs usually integrate two or more, or even tens of cores on a single chip to realize a performance leap through parallel processing. For example, the current top-end Intel server processor, the Xeon E7-8894 v4, is equipped with 24 CPU cores with a base clock speed of 2.4 GHz, and a 60 MB cache.

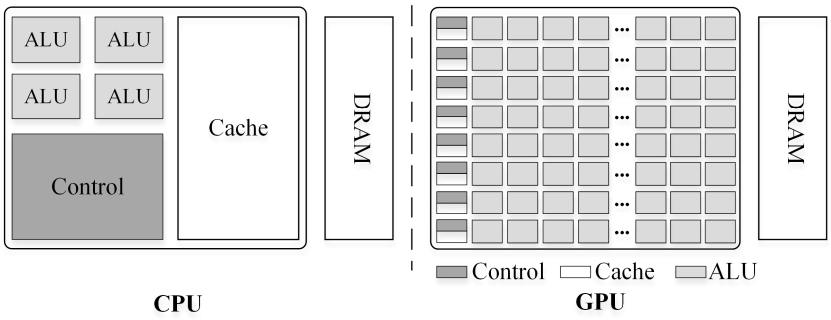


Figure 1. Different architectures of a CPU and GPU.

In contrast, a many-core GPU is comprised of hundreds, even thousands, of stream processors, which share common control units and execute instructions in order. Within a GPU, most of the transistors are used for computation rather than complex instruction-level parallelism and large caches. Although GPU cores run at a lower frequency and use smaller-sized caches [13], they are able to issue thousands of concurrent threads, and thus hide memory latency and achieve massive parallelism. Therefore, a many-core GPU is well-suited for data-parallel and high throughput computation.

Multi-threaded programming for multicore CPUs has been studied extensively, and many programming languages and software libraries have been developed. Typical programming languages/libraries include Pthreads, OpenMP, Threading Building Blocks (TBB), and the Math Kernel Library (MKL). Among them, Pthreads is a low-level API (Application Program Interface) and provides basic thread operations, such as creation, destruction, and synchronization. OpenMP has been widely used and is the de facto standard for shared-memory parallel programming due to its portability and ease of programming. TBB and MKL were both launched by the Intel Corporation and were specially optimized for Intel multicore processors. TBB is an open source C++ template library for parallel programming, while MKL contains many optimized math routines for scientific computation.

For many-core GPUs, typical programming languages include OpenCL, Brook+ and CUDA. OpenCL (Open Computing Language) is a recent standard, which is ratified by the Khronos Group, for cross-platform parallel programming with diverse processors [14]. OpenCL is welcomed for its portability, but it cannot achieve the highest possible performance for its high-level abstraction [15]. Brook [16] is an extension to the C-language for stream programming that was originally developed by Stanford University; Brook+ is an implementation of the Brook GPU specification on AMD's compute abstraction layer. CUDA (Compute Unified Device Architecture) [12] is a programming model that was created by the NVIDIA company for general purpose processing on CUDA-enabled GPUs. Among them, CUDA is the most widely used because of its ease of programming and much bigger market share of NVIDIA GPUs. CUDA is also superior for its mature ecosystem. Many GPU-accelerated libraries have been developed along with CUDA, e.g., cuBLAS, Thrust, Magma, cuFFT.

2.2. Heterogeneous Computing

Recently high performance computing has increasingly turned to CPU-GPU heterogeneous systems to fulfill super large-scale scientific and engineering computations. A large proportion of global supercomputers have adopted the heterogeneous CPU-GPU architecture from the TOP500 and Green500 websites. Furthermore, commodity computers, servers, and workstations are also typical heterogeneous CPU-GPU platforms. On these ubiquitous platforms, heterogeneous computing arises and aims to exploit all of the available resources from different types of PUs thus achieving

better performance gains. However, due to the dramatically different architectures, programming models, and computational capabilities of the CPU and GPU, the following challenges are involved in heterogeneous computing.

The lack of uniform programming techniques is the first challenge. Current CPU programming languages and software libraries cannot be directly ported for GPU application development. Although OpenCL offers a common API for task-parallel and data-parallel heterogeneous computing, and has been used in many fields [17–19], OpenCL applications usually suffer from performance loss due to its high-level programming abstraction. Brook+ and CUDA only work for their own compatible GPUs. Thus, no uniform programming language can fully exploit the potential of a heterogeneous CPU-GPU system. Hybrid-programming models are usually chosen to realize the highest possible performance, e.g., OpenMP + CUDA [20]. On the CPU side, leveraging typical programming languages and software libraries can easily realize the efficient utilization of multicore CPUs. On the GPU side, CUDA is a good choice for its ease of programming and ecosystem maturity. The disadvantage of a hybrid-programming model is that it exerts additional learning burden to users.

The second challenge is inefficient CPU-GPU collaboration. Currently in heterogeneous CPU-GPU systems, the multicore CPU is conventionally responsible for task scheduling and I/O control, while the GPU is burdened with all of the computation. For example, recent research on spatial interpolation [21–23] have efficiently exploited the power of GPUs and other accelerators, but left the parallel computing resources of multicore CPU underutilized. In heterogeneous computing, multicore CPUs must not only work as hosts, but also offload computation from GPUs so that closer collaboration and better performance can be achieved.

Task scheduling is the third challenge. A scheduling strategy should consider both the internal characteristics of target algorithms and the external hardware attributes of the underlying PUs to determine suitable task partitioning and allocation. Currently, Many research focuses on algorithm-level workload partitioning and scheduling [15]. Workload partitioning techniques have been designed based on the relative performance of PUs [20,24], the nature of subtasks [25], or other partitioning criteria for different algorithms and applications. As for scheduling, dynamic and static scheduling policies have been extensively studied in order to maintain workload balance. However, task scheduling still remains a challenge since empirical analyses are always needed to design proper workload partitioning methods and scheduling policies for specific scenarios.

3. Thin Plate Spline Interpolation

3.1. TPS Introduction

TPS is a spline-based SI method that can preserve the derived interpolation surface as smoothly as possible. Here, a short introduction to two-dimensional (2D) TPS interpolation is presented.

Given a set of three-dimensional (3D) sample points $\{P_i = (x_i, y_i, z_i) | i = 1, \ldots, n\}$, (x_i, y_i) denotes the 2D space coordinates of the ith sample point with an associated sampling value z_i. The TPS algorithm aims to seek a smoothness function F that passes through each sample point exactly while minimizing the "bending energy". The case of 2D bending energy is defined as

$$I(F) = \iint_{R^2} (F''^2_{xx} + 2F''^2_{xy} + F''^2_{yy}) dxdy \tag{1}$$

Further, it has been shown [26] that the problem of minimizing Equation (1) is subject to the constraints in Equation (2)

$$\sum_{i=1}^{n} \lambda_i = 0, \sum_{i=1}^{n} \lambda_i x_i = 0, \sum_{i=1}^{n} \lambda_i y_i = 0, \tag{2}$$

where λ_i is the coefficient for the ith spline term. Thus, we derive a TPS interpolation function that satisfies

$$\begin{aligned} F(x,y) &= a_0 + a_1 x + a_2 y + \sum_{i=1}^{n} \lambda_i \phi(\|(x,y) - (x_i,y_i)\|) \\ &= a_0 + a_1 x + a_2 y + \sum_{i=1}^{n} \lambda_i r_i^2 \ln r_i \end{aligned} \tag{3}$$

In Equation (3), we define ϕ as $\phi(r) = r^2 \ln r$, which is in fact a radial basis function. The $||\cdot||$ denotes the Euclidean distance between the interpolation point and each sample point, i.e., $r_i = \sqrt{(x-x_i)^2 + (y-y_i)^2}$, and a_0, a_1, a_2 are the coefficients of the planar term of the spline.

Therefore, TPS interpolation contains two steps, i.e., coefficient solution and z-value estimation. With Equations (2) and (3), the coefficient solution can be represented in the form of a linear system as in Equation (4)

$$\begin{pmatrix} 0 & \dots & \varphi(r_{1n}) & 1 & x_1 & y_1 \\ \varphi(r_{21}) & \dots & \varphi(r_{2n}) & 1 & x_2 & y_2 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \varphi(r_{n1}) & \dots & 0 & 1 & x_n & y_n \\ 1 & \dots & 1 & 0 & 0 & 0 \\ x_1 & \dots & x_n & 0 & 0 & 0 \\ y_1 & \dots & y_n & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \dots \\ \lambda_n \\ a_0 \\ a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} z_1 \\ z_2 \\ \dots \\ z_n \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{4}$$

The z-value for each interpolation point can be estimated with the derived function F as in Equation (3). The computational complexity of the two steps is $O(n^3)$ and $O(n \cdot p)$, where n and p denote the number of the sample points and interpolation points, respectively.

3.2. Local TPS Interpolation

In global TPS interpolation, with the increase of sample points, the linear system in Equation (4) is prone to ill-condition that will result in incorrect coefficients. Direct computing without matrix transformation is only suitable for a very small n [27]. Unfortunately, the number of points in massive LiDAR point clouds can easily reach millions, or even billions. Global TPS interpolation therefore becomes unfeasible and extremely time-consuming.

Thus, to overcome this problem, TPS interpolation in this paper was implemented as a local interpolation. Local interpolation only requires a few tens of neighboring sample points to derive an estimation value. Figure 2 displays the main steps of using local TPS interpolation for massive LiDAR point clouds. The input point clouds are firstly read and divided into a collection of discrete data blocks so that they are able to fit into main memory and graphic memory. A grid index is then constructed for each block in order to reduce the time spent in the nearest neighbor search for each interpolation point. Next, local TPS interpolation, including nearest neighbor search, coefficient solution, and z-value estimation can be implemented point by point within each block. After interpolating all of the blocks, a raster Digital Elevation Model (DEM) will be generated. Throughout the process, local TPS interpolation is the most time-consuming step. Interpolation among local points and that among different blocks are both independent and ideal for parallelization.

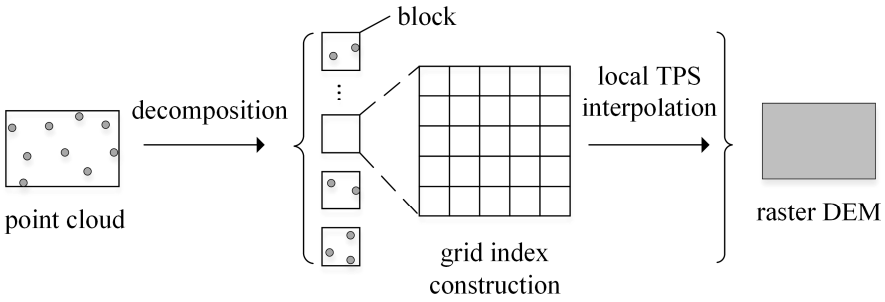


Figure 2. Main steps of using local Thin Plate Spline (TPS) interpolation for LiDAR point clouds.

4. Framework of the Hybrid Parallel Interpolation

4.1. Introduction of the Interpolation Framework

As shown in Figure 3, a parallel framework based on a hybrid-programming model (Pthreads + Intel TBB + CUDA) was designed to accelerate TPS interpolation from massive LiDAR point clouds. This parallel interpolation framework contains four stages: fast online training, streaming decomposition, task scheduling, and parallel interpolation. The first stage quickly finds the optimal decomposition granularity so that each PU is able to run at maximum speed. Based on the derived granularity, raw point clouds are then decomposed into a stream of discrete blocks with a streaming decomposition method. In the third stage, a dynamic scheduler called HDSG is proposed to assign blocks and computation to each PU so that parallel TPS interpolation can be executed in the fourth stage.

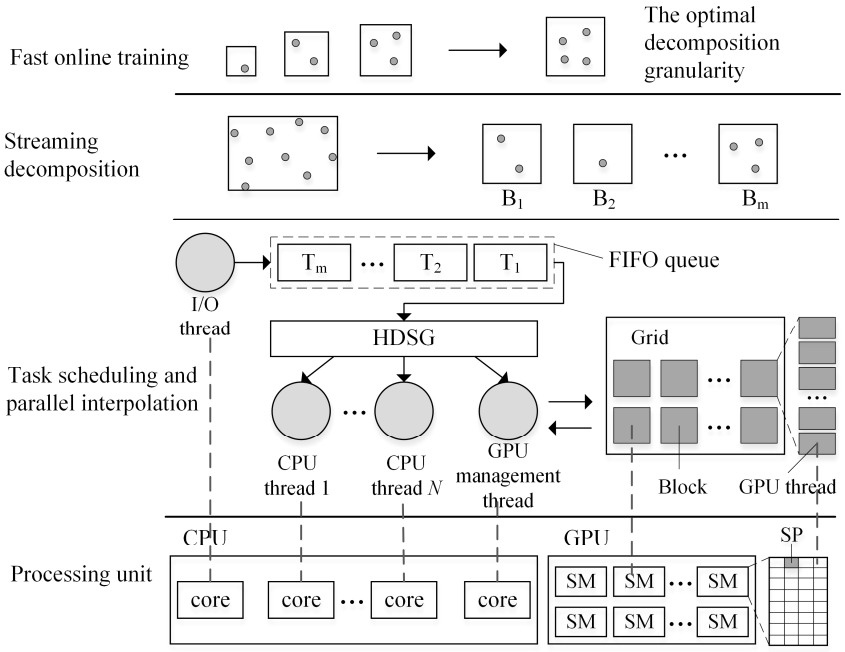


Figure 3. The framework of the hybrid parallel interpolation.

Once a block is ready, it is delivered to a First-In-First-Out (FIFO) task queue. This task queue is shared by CPU threads and supports concurrent accesses. Two operations are supported on this queue: task submission (push), and request for a task to execute (pop). In implementation, more than one task queue is generated and used for data storage and task scheduling.

In this framework, N + 2 CPU threads are generated. One I/O thread is responsible for data reading and decomposition, and another thread is dedicated to GPU management, including device

initialization, data transfer, and kernel launching. The other N CPU threads are assigned to do TPS interpolation in parallel. The multicore CPU can process N blocks simultaneously, while the GPU handles only one block at a time. Since interpolation of each point is independent in local TPS, it is executed in a GPU thread as the smallest work unit. By launching a massive number of GPU threads simultaneously, the parallel computing power of the GPU can be fully exploited.

4.2. Fast Online Training

In a heterogeneous CPU-GPU system, as the computing capability of the CPU and GPU are much different, block size becomes a key factor to achieve a balanced workload and efficient resource utilization. GPU performance is much more sensitive to block size than a CPU given the massive-threaded parallelism of a GPU. If the block size is too small, thread initialization and data transfer between the CPU and GPU will take most of the time, thus leading to GPU underutilization. On the other hand, larger blocks can achieve better GPU performance, but may result in a severe workload imbalance between the CPU and GPU.

To find the optimal decomposition granularity, a series of experiments were conducted to model the quantitative relationship between block size and GPU processing speed. Sample data consisting of four point clouds at different point densities were chosen for the decomposition experiments implemented on an NVIDIA Fermi C2050 GPU. According to empirical research, only about 10–30 neighboring points are required to interpolate each point [28]. In these experiments, the number of neighboring points requested for local interpolation was set to 15, and the radius of neighbor search and interpolation resolution were set to 15 m and one meter, respectively.

In Figure 4, block sizes are represented by the number of pixels (i.e., interpolation points), while GPU processing speed is measured in pixels per second. Based on the distribution of scatter points in Figure 4, we infer that GPU processing speed over block size obeys a logarithmic curve. When the block size is very small, the GPU is underutilized due to lack of enough GPU threads. The interpolation speed increases rapidly with block size, finally evolving towards stability as the block size exceeds the number of CUDA cores inside the GPU. This trend illustrates that the massive-threaded parallelism within the GPU was gradually exploited. For validation, R square was used to measure the confidence of fit curves. Figure 4 shows that the R square values for different LiDAR point clouds were very high, ranging from 0.9256 to 0.9709. A higher value for R square indicates a better fit. Therefore, the curve fit results validate the assumption that GPU processing speed over block size indeed obeys a logarithmic curve.

Hence, the optimal block size u_0 (in pixels) can be predicted with a logarithmic equation

$$v = a \ln(u) + b \tag{5}$$

where u and v denote the number of pixels in a block and GPU processing speed, respectively. In order to reach convergence quickly, least squares estimation is used in online training to fit the logarithmic curve with as few samples as possible (e.g., four block sizes). As long as the parameters a, b are calculated, the slope of the fitted logarithmic curve is used to measure whether the GPU processing speed reaches stability. When the slope decreases to a reasonable threshold, which is empirically set as 0.01, the GPU processing speed will be treated as stable. Thus, the optimal block size u_0 is obtained and the optimal decomposition granularity d (in meters) can be calculated according to

$$d = r * \sqrt{u_0} \tag{6}$$

where r denotes the raster resolution for spatial interpolation. The online training method is fast and timesaving because the logarithmic curve fitting is rapid and the calculation of d is simple.

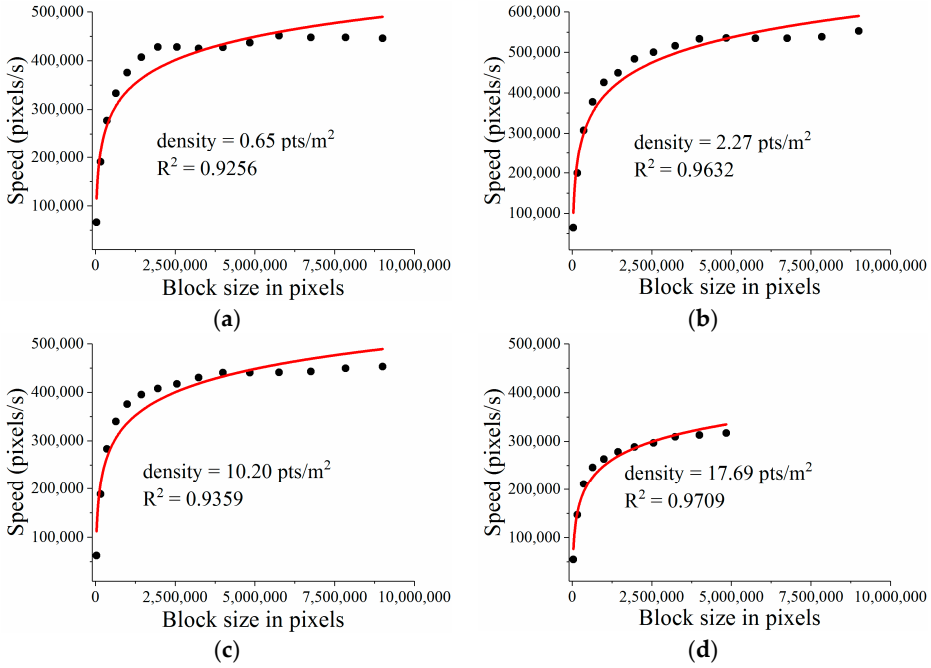


Figure 4. GPU processing speed (pixels per second) for different block sizes. (**a**) sample data 1; (**b**) sample data 2; (**c**) sample data 3; (**d**) sample data 4.

4.3. Streaming Decomposition

After calculating the optimal decomposition granularity d, the input point clouds are split into discrete blocks so that all of the blocks can be dispatched to different PUs for parallel interpolation. Based on d and the overlap width w, a regular square block decomposition strategy is adopted. The overlap width w is used to guarantee the same nearest neighbor search results on the edge of each block and preserve the smoothness and continuity of the derived raster surface. Therefore, the overlap width w should be no smaller than the neighbor search radius. Since the data volume of LiDAR point clouds can be tens or even hundreds of times of the memory capacity, a streaming decomposition method is implemented to continuously generate discrete blocks.

Shown in Figure 5, the streaming decomposition method reads LiDAR points twice. The decomposition grid frame is established from the obtained block width d and the boundary rectangle of input data. A first read pass is applied over all of the input points to count the number of points within each block. These counts are later used to identify whether blocks are full in the second read pass and therefore ready to pipe into FIFO task queues. Due to the streaming decomposition, at any given time only a small fraction of the raw point clouds resides in the main memory, while the remainder still resides on the hard drive.

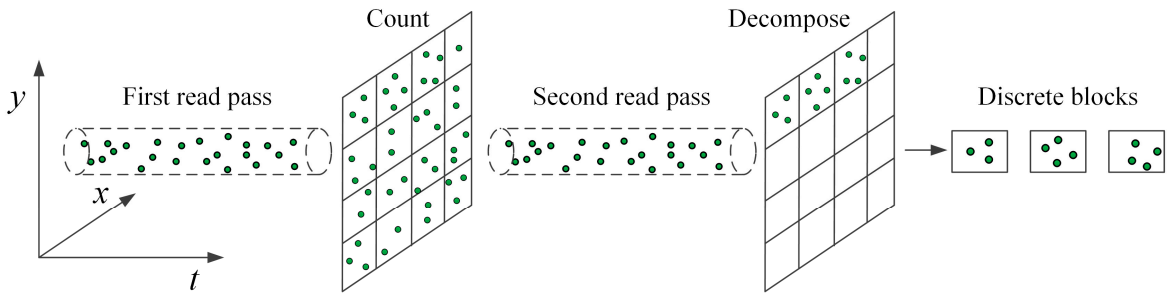


Figure 5. Streaming decomposition of input point clouds.

4.4. HDSG Scheduling Strategy

Due to the irregular shape of input point clouds (illustrated in Figure 6), square block decomposition generates many edge blocks that are much smaller and contain fewer points than the other blocks. These blocks, if allocated to GPU, will cause resource underutilization. At the same time, fixed block size partitioning is prone to load imbalance due to the huge performance gap between the CPU and GPU. For example, when interpolation ends, the GPU is very likely to be idle and waits for slow CPU cores to finish the remaining tasks.

For fixed block size partitioning, the two best available dynamic scheduling strategies are the greedy policy and work stealing. In the greedy policy, idle workers (i.e., computation resources) actively request tasks from a central queue to process. Work stealing creates per-worker queues instead of a central queue. If a worker becomes idle, it steals a task from the worker with the heaviest load. These two scheduling strategies are easy to implement, but they cannot address the differences in CPU/GPU computing capabilities. Thus, HDSG, a heterogeneous dynamic scheduler based on the greedy policy, is proposed with two extensions—task priority declaration and block resizing.

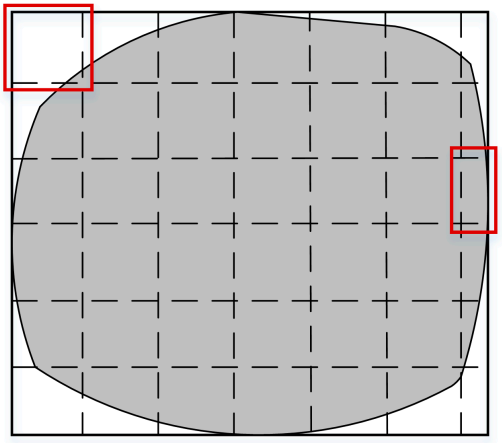


Figure 6. Edge blocks with slight computation burden.

As shown in Figure 7, the generated blocks are divided into two groups based on the computational weight, which represents the computation burden of each block. The computational weight of each block is expressed as

$$W_i = \frac{D_i}{D_{avg}} \times \frac{S_i}{S_{std}} \tag{7}$$

where D_i, S_i are the average point density and the block size (in pixels) of the ith block, respectively; S_{std} denotes the optimal block size u_0; D_{avg} represents the global average point density; and, W_{std} is a threshold set empirically as 0.1. Blocks with larger computational weight ($W_i > W_{std}$) are pushed into a task queue called *cg_queue*, and the rest of the blocks that may lead to GPU underutilization will be delivered into *c_queue* and processed only by the CPU. Each task queue will be given at least one priority level, represented by a string (e.g., "C1"). Taking the *cg_queue* as an example, it has two priority levels—"G1" and "C3", meaning that both the GPU and CPU have access to the *cg_queue*; and processing blocks in the *cg_queue* is the highest priority for the GPU but only the third highest priority for the CPU.

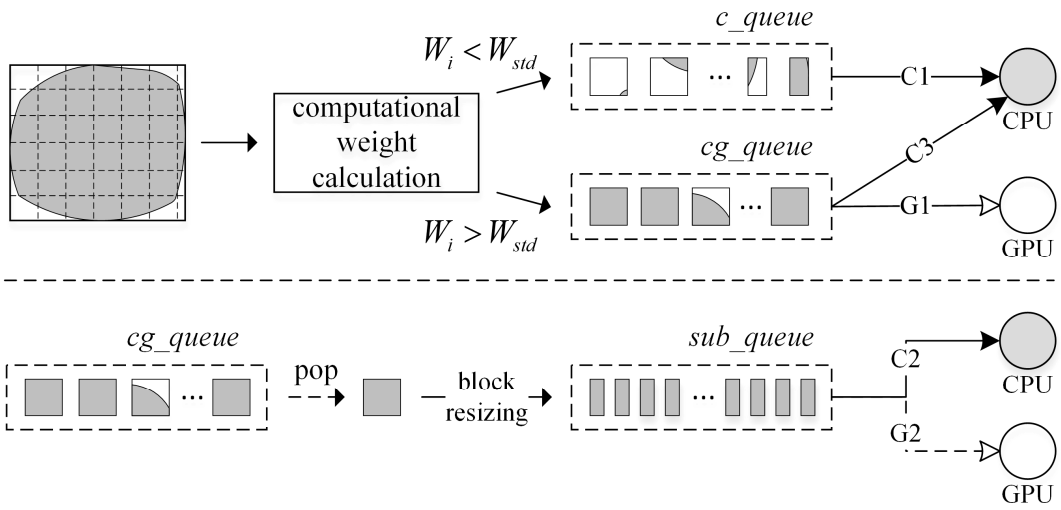


Figure 7. Illustration of the Heterogeneous Dynamic Scheduler based on the Greedy policy (HDSG) scheduling strategy.

Since the CPU is less sensitive to block sizes, block resizing is adopted for the CPU to ensure better load balancing. Blocks in the *cg_queue* can be popped and further divided into a collection of sub-blocks. These sub-blocks are stored in another task queue called *sub_queue*, which is shared by the CPU and GPU. Block resizing can be implemented at any time during the interpolation, and as long as the *sub_queue* is not empty, the CPU will request blocks from the *sub_queue* instead of the *cg_queue*. GPU may turn to the *sub_queue* for blocks near the end of interpolation if the *cg_queue* becomes empty. Block resizing is realized based on the average interpolation speed of GPU (V_{GPU}) and CPU (V_{CPU}), which is tracked and updated throughout the execution. A block in the *cg_queue* is divided into a collection of stripes horizontally/vertically on the condition that the number of stripes is equal to V_{GPU}/V_{CPU}. Notably, the size of *sub_queue* is set to a small value (e.g., N) so as not to generate too many sub-blocks, which would reduce GPU utilization near the end of completion.

5. Experiments and Discussion

5.1. Design and Configuration

Two performance evaluation experiments were conducted. The first validated the effectiveness of online training. The second experiment evaluated the overall performance of the hybrid parallel framework and the scheduling efficiency of our HDSG policy. The proposed method was compared with the greedy policy and work stealing.

These experiments were carried out on two high-performance computers. The first computer (named hpc_01) was equipped with two 6-core Intel Xeon E5-2620 CPUs (i.e., 12 cores total) and an NVIDIA Tesla C2050 GPU, while the second one (named hpc_02) was equipped with an Intel Core i7-6850K CPU and an NVIDIA GTX 1060 GPU. Detailed information for each PU is listed in Tables 1 and 2.

Table 1. Detailed information of experimental Processing Units (Pus) on hpc_01.

PU Name	Intel Xeon E5-2620	NVIDIA Tesla C2050
Frequency	2.0 GHz	1.15 GHz
Number of cores	6 cores	448 CUDA cores
Memory bandwidth	42.6 GB/s	144 GB/s
Double-precision gigaflops	96.0 GFLOPS	515.2 GFLOPS

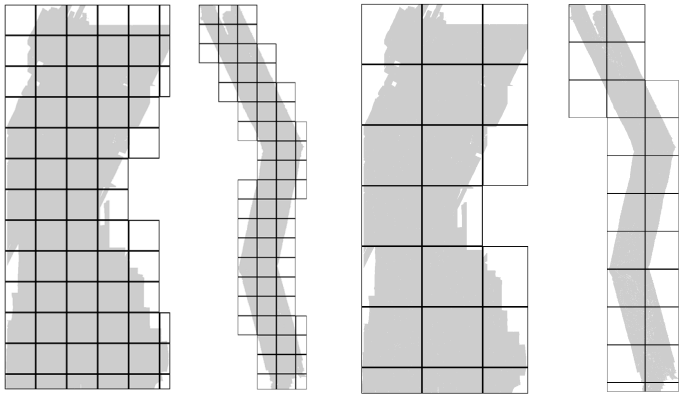
Table 2. Detailed information of experimental PUs on hpc_02.

PU Name	Intel Core i7-6850K	NVIDIA GTX 1060
Frequency	3.6 GHz	1.71 GHz
Number of cores	6 cores	1152 CUDA cores
Memory bandwidth	76.8 GB/s	192.2 GB/s
Double-precision gigaflops	345.6 GFLOPS	1967.6 GFLOPS

Two experimental LiDAR point clouds (listed in Table 3) were downloaded from the OpenTopography website (www.opentopography.org). Figure 8 shows the general shapes and decomposed blocks from the point cloud data. The irregular outline of *pcd_02* resulted in more blocks with slight computation burden when compared with *pcd_01*. Such a difference is helpful when verifying the effectiveness of our HDSG policy. Fifteen neighboring points were searched for local interpolation, and the interpolation resolution was set to one meter. The search radius was set to 15 m, which is reasonably large so that enough neighboring points could be searched for each interpolation point.

Table 3. Basic information about experimental LiDAR point clouds.

Data Name	Point Density	Area	Source	File Size
pcd_01	2.62 pts/m^2	365 km^2	Airborne Lidar	24.9 GB
pcd_02	3.61 pts/m^2	371 km^2	Airborne Lidar	34.9 GB



(**a**) (**b**)

Figure 8. Grid decomposition of different LiDAR point clouds. (**a**) *pcd_01* and *pcd_02* on hpc_01; (**b**) *pcd_01* and *pcd_02* on hpc_02.

5.2. Online Training Efficiency Experiment

Online training was performed repeatedly on different point clouds to evaluate the time efficiency. The training datasets were clipped from the original data and only covered a small region. The time spent in training, reading, and decomposition was recorded as in Table 4. The time spent in reading data and decomposition was recorded from the start of reading to the moment when the first block was sent to the task queue. As shown in Table 4, the ratio between online training and reading and decomposition was less than 3%. Therefore, the online training quickly finds the optimal decomposition granularity, and the training time can be neglected.

Table 4. Training time and the optimal decomposition granularities of different datasets.

Platform	Data Name	Training	The Optimal Granularity	Data Reading & Decomposition
hpc_01	*pcd_01*	3.7 s	3040 m	131.8 s
	pcd_02	4.2 s	3030 m	166.5 s
hpc_02	*pcd_01*	3.7 s	5910 m	155.2 s
	pcd_02	4.2 s	5860 m	184.7 s

To validate that the GPU runs at maximum speed with the optimal decomposition granularity d, an interpolation with different block sizes was carried out with only GPU acceleration. The experimental data was randomly clipped from the original data. As shown in Figure 9, the interpolation time reduced sharply with an increase in block size, and became stable when the block size was near or over d, that is, the interpolation time under the block size d was almost the shortest and showed little improvement over larger block sizes. Therefore, the online training method with a logarithmic curve fit is valid and cost-effective when finding the optimal decomposition granularity that maximizes GPU utilization.

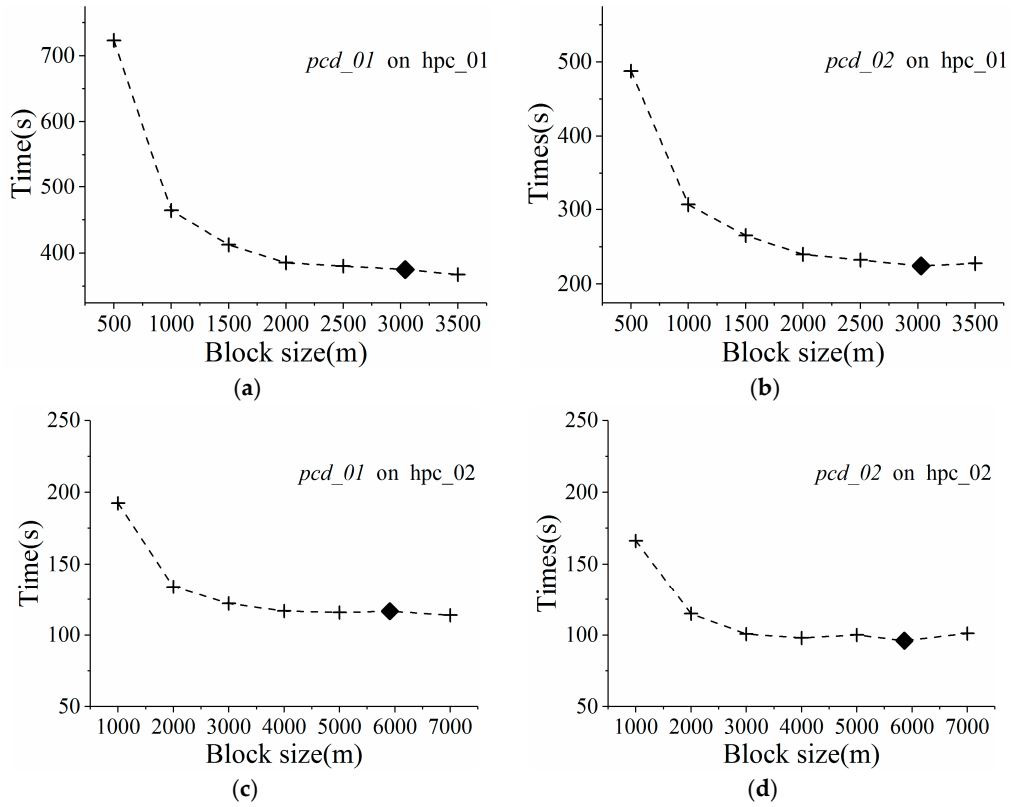


Figure 9. GPU interpolation time under different decomposition granularities. (**a**) *pcd_01* on hpc_01; (**b**) *pcd_02* on hpc_01; (**c**) *pcd_01* on hpc_02; (**d**) *pcd_02* on hpc_02.

5.3. Parallel TPS Interpolation Experiment

The second experiment evaluated the speedup of the hybrid parallel TPS interpolation algorithm in comparison with the sequential mode and other parallel modes that only utilize either the CPU or GPU. Furthermore, different task scheduling strategies were also used to measure the efficiency of the proposed HDSG policy. Tables 5 and 6 show the interpolation time and corresponding speedups of different execution modes on hpc_01 and hpc_02. For the GPU-only mode and the hybrid (CPU-GPU)

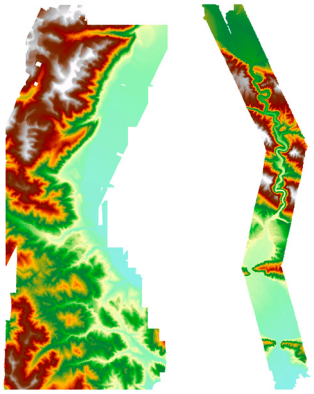
mode, the interpolation time refers to the total time spent in TPS interpolation and data delivery between the CPU and GPU. The derived raster DEMs are shown in Figure 10.

Table 5. Interpolation time (s) and speedups of different execution modes on hpc_01.

Execution Mode	*pcd_01*		*pcd_02*	
	Time (s)	**Speedup**	**Time (s)**	**Speedup**
CPU (serial)	11038.9	—	8453.8	—
CPU (parallel)	1268.7	8.7	975.7	8.6
GPU	876.1	12.6	661.4	12.7
CPU-GPU (HDSG)	546.1	20.2	459.9	18.3

Table 6. Interpolation time (s) and speedups of different execution modes on hpc_02.

Execution Mode	*pcd_01*		*pcd_02*	
	Time (s)	**Speedup**	**Time (s)**	**Speedup**
CPU (serial)	3348.5	—	2673.4	—
CPU (parallel)	496.3	6.7	473.8	5.6
GPU	263.2	12.7	261.4	10.2
CPU-GPU (HDSG)	189.5	17.7	176.1	15.2



(**a**) (**b**)

Figure 10. Raster Digital Elevation Model (DEM) interpolated by the hybrid parallel TPS algorithm. (**a**) *pcd_01*; (**b**) *pcd_02*.

Our hybrid parallel algorithm delivered a significant reduction in interpolation time, not only from the serial CPU mode but also from the CPU-only and GPU-only parallel modes. The hybrid parallel mode also showed a considerable jump in performance speedup on both experimental platforms. For example, the speedup S exceeded 20 in *pcd_01* on hpc_01, while the corresponding speedups of the CPU-only mode and GPU-only mode were 8.7 and 12.6, respectively. On hpc_02, although the theoretical performance gap between the multicore CPU and GPU is much larger than that on hpc_01, the hybrid parallel algorithm still achieved better performance gain as compared to the GPU-only mode.

We use a max speedup S_{max} to evaluate if the hybrid parallel mode fully exploits all of the available parallel resources in a heterogeneous CPU-GPU system. Assuming that computation within each block is equal, the parallel interpolation time T_{cg} in a heterogeneous CPU-GPU system can be represented by

$$T_{cg} = \max\left(\frac{m_c}{m}T_c, \frac{m - m_c}{m}T_g\right), \tag{8}$$

where T_c, T_g represents the parallel interpolation time using either the CPU or GPU; m denotes the total number of generated blocks, and m_c represents the number of blocks interpolated by the CPU. The minimum of T_{cg} in Equation (7) is determined only if the interpolation time of CPU is equal to that of GPU. Thus, m_c and T_{cg-min} can be calculated by

$$\begin{aligned} T_{cg-\min} &= \frac{m_c}{m} * T_c = \frac{m-m_c}{m} * T_g \Rightarrow m_c = \frac{m*T_g}{T_c+T_g} \\ \Rightarrow T_{cg-\min} &= \frac{T_c*T_g}{T_c+T_g} \end{aligned} \tag{9}$$

According to Amdahl's Law, the max speedup S_{max} achieved in a heterogeneous CPU-GPU system is defined as

$$S_{max} = \frac{T_s}{T_{cg-min}} = \frac{T_s}{\frac{T_c*T_g}{T_c+T_g}} = \frac{T_s}{T_c} + \frac{T_s}{T_g}, \tag{10}$$

where T_s is the serial interpolation time using only the CPU. With Equation (9), the max speedup S_{max} for *pcd_01* and *pcd_02* can be calculated. As shown in Figure 11, the difference between the actual speedup S and S_{max} was small, e.g., 1.1 for *pcd_01* on hpc_01. For *pcd_02* on hpc_01, the speedup difference was a little bigger (i.e., 3.0). This bigger difference could be attributed to the irregular shape of point clouds, which led to load imbalances near the end of the interpolation. Therefore, the comparison of the actual speedup S with S_{max} demonstrates that the hybrid parallel mode can efficiently exploits all of the available computing resources in a heterogeneous CPU-GPU system.

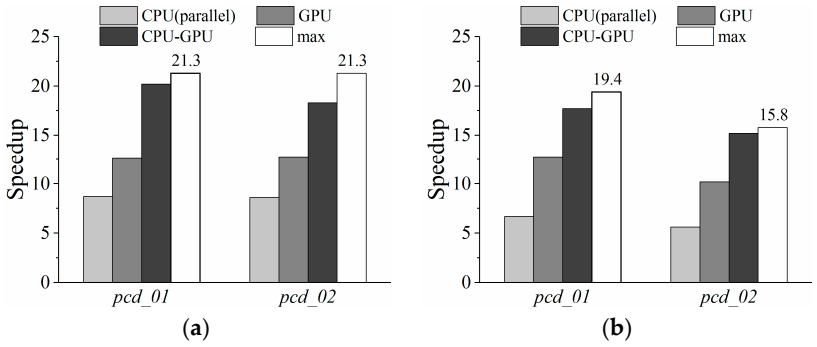


Figure 11. Speedups of different parallel execution modes for different point clouds. (**a**) hpc_01; (**b**) hpc_02.

For further evaluating the efficiency of HDSG policy, two common dynamic scheduling strategies were also implemented for comparative purposes, i.e., the greedy policy and work stealing. The interpolation time and speedups of the hybrid parallel algorithm with different scheduling strategies are displayed in Tables 7 and 8 and Figure 12. When compared to the greedy policy and work stealing, the HDSG policy showed considerable load balancing improvement. On hpc_01, about 12% and more than 20% of the total interpolation time were reduced for *pcd_01* and *pcd_02*. The improvement for *pcd_02* was much bigger since blocks with slight computation burden accounted for a larger proportion. By declaring priorities, the HDSG policy demonstrates a capability for improving GPU utilization when there are a large percentage of blocks with a slight computation burden. On hpc_02, attributing to the huge performance gap between the CPU and GPU, the hybrid interpolation algorithm using greedy policy or work stealing was even slower than the GPU-only mode. Given that very few blocks were pushed into the *c_queue*, declaring priority made little contribution to load balancing on hpc_02. By implementing block subdivision, load imbalance between the CPU and GPU was greatly alleviated. Therefore, the HDSG policy designed for heterogeneous CPU-GPU systems can effectively alleviate load imbalances.

Table 7. Interpolation time (s) and speedups of the hybrid parallel algorithm with different scheduling strategies on hpc_01.

Scheduling Strategy	*pcd_01*		*pcd_02*	
	Time (s)	**Speedup**	**Time (s)**	**Speedup**
greedy policy	621.8	17.7	580.4	14.5
work stealing	622.0	17.7	575.0	14.7
HDSG (no priority)	557.9	19.7	497.1	17.0
HDSG	546.1	20.2	459.9	18.3

Table 8. Interpolation time (s) and speedups of the hybrid parallel algorithm with different scheduling strategies on hpc_02.

Scheduling Strategy	*pcd_01*		*pcd_02*	
	Time (s)	**Speedup**	**Time (s)**	**Speedup**
greedy policy	289.8	11.6	265.0	10.1
work stealing	281.0	11.9	281.2	9.5
HDSG (no priority)	189.0	17.7	179.0	14.9
HDSG	189.5	17.7	176.1	15.2

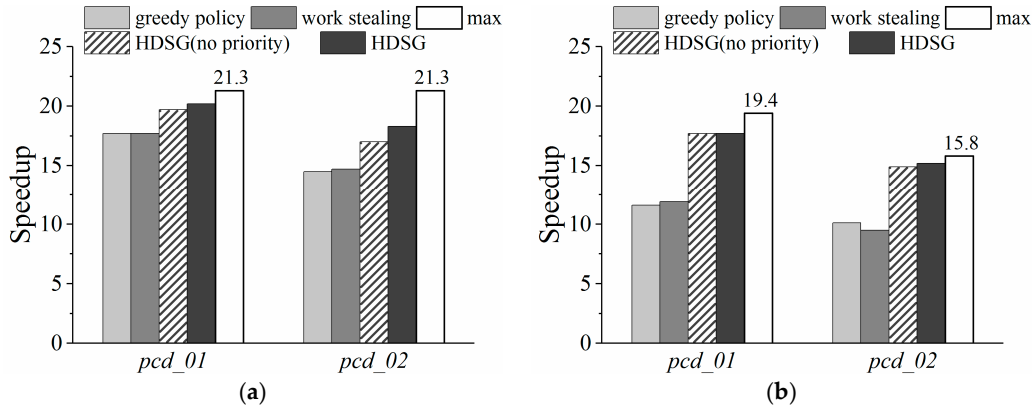


Figure 12. Speedups of the hybrid parallel algorithm with different scheduling strategies. (**a**) hpc_01; (**b**) hpc_02.

6. Conclusions and Future Work

Nowadays, CPU-GPU computing platforms are available everywhere, from commercial personal computers to dedicated powerful workstations. The emergent challenge for researchers is how to design and implement efficient parallel algorithms to harvest all of the available heterogeneous parallel resources. In this paper, a TPS interpolation for massive point clouds was explored to address this challenge in the integration of multicore CPUs and many-core GPUs.

A simple yet powerful SI framework was designed to leverage the power of both the CPU and GPU. Using a logarithmic function model, optimum decomposition granularity can be rapidly calculated to maximize the GPU utilization. Our HDSG scheduling strategy keeps the heterogeneous system fully loaded and alleviates workload imbalance by priority declaration and block resizing. Experiments on high-end workstations demonstrate the efficient exploitation of heterogeneous parallel resources.

In the future, this hybrid parallel SI framework will integrate more local spatial interpolation algorithms, such as IDW and Kriging, to make it more feasible for practical use. When faced with even larger volumes of point clouds, this framework can be extended to accommodate heterogeneous CPU-GPU clusters for further improvement in overall performance.

Acknowledgments: This work is supported by National Key R&D Program of China (2017YFB0503801) and the Natural Science Foundation of China (Grant No. 41301411).

Author Contributions: Hongyan Wang and Xuefeng Guan conceived and designed the algorithm and experiments; Hongyan Wang performed the experiments; all authors analyzed the data and experimental results; Huayi Wu contributed the high-performance computing infrastructure and gave other financial aid; and Hongyan Wang and Xuefeng Guan wrote the paper. In addition, we sincerely thank Steve McClure for the language polishing and revising.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

PU	processing unit
SI	spatial interpolation
TPS	thin plate spline
DEM	Digital Elevation Model
HDSG	heterogeneous dynamic scheduler based on the greedy policy

References

1. Lam, N.S.-N. Spatial interpolation methods: A review. *Cartogr. Geogr. Inf. Sci.* **1983**, *10*, 129–150. [CrossRef]
2. Jaboyedoff, M.; Oppikofer, T.; Abella´n, A.; Derron, M.-H.; Loye, A.; Metzger, R.; Pedrazzini, A. Use of lidar in landslide investigations: A review. *Nat. Hazards* **2012**, *61*, 5–28. [CrossRef]
3. Armstrong, M.P.; Marciano, R. Massively parallel processing of spatial statistics. *Int. J. Geogr. Inf. Syst.* **1995**, *9*, 169–189. [CrossRef]
4. Gajraj, A.; Joubert, W.; Jones, J. *A Parallel Implementation of Kriging with a Trend*; Technical Report; Los Alamos National Laboratory: Los Alamos, NM, USA, 1 November 1997; Volume 22, pp. 239–261.
5. Kerry, K.E.; Hawick, K.A. Kriging interpolation on high-performance computers. In Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking, Amsterdam, The Netherlands, 21–23 April 1998; Springer: Berlin/Heidelberg, Germany; pp. 429–438.
6. Hawick, K.A.; Coddington, P.D.; James, H.A. Distributed frameworks and parallel algorithms for processing large-scale geographic data. *Parallel Comput.* **2003**, *29*, 1297–1333. [CrossRef]
7. Cheng, T.; Li, D.; Wang, Q. On parallelizing universal kriging interpolation based on openmp. In Proceedings of the 2010 Ninth International Symposium on Distributed Computing and Applications to Business Engineering and Science (DCABES), Hong Kong, China, 10–12 August 2010; pp. 36–39.
8. Guan, X.; Wu, H. Leveraging the power of multi-core platforms for large-scale geospatial data processing: Exemplified by generating dem from massive lidar point clouds. *Comput. Geosci.* **2010**, *36*, 1276–1282. [CrossRef]
9. Ravé, E.G.; Jiménez-Hornero, F.J.; Ariza-Villaverde, A.B.; Gómez-López, J.M. Using general-purpose computing on graphics processing units (GPGPU) to accelerate the ordinary kriging algorithm. *Comput. Geosci.* **2014**, *64*, 1–6. [CrossRef]
10. Cheng, T. Accelerating universal kriging interpolation algorithm using CUDA-enabled GPU. *Comput. Geosci.* **2013**, *54*, 178–183. [CrossRef]
11. Beutel, A.; Mølhave, T.; Agarwal, P.K. Natural neighbor interpolation based grid dem construction using a gpu. In Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems, San Jose, CA, USA, 2–5 November 2010; pp. 172–181.
12. NVIDIA Corporation. Cuda C Programming Guide. Available online: http://docs.nvidia.com/cuda/cuda-c-programming-guide/ (accessed on 15 September 2017).
13. Mittal, S. A survey of techniques for managing and leveraging caches in GPUS. *J. Circuits Syst. Comput.* **2014**, *23*, 229–236. [CrossRef]
14. Stone, J.E.; Gohara, D.; Shi, G. Opencl: A parallel programming standard for heterogeneous computing systems. *Comput. Sci. Eng.* **2010**, *12*, 66–72. [CrossRef] [PubMed]
15. Mittal, S.; Vetter, J.S. A survey of cpu-gpu heterogeneous computing techniques. *ACM Comput. Surv.* **2015**, *47*, 1–35. [CrossRef]

16. Buck, I.; Foley, T.; Horn, D.; Sugerman, J.; Fatahalian, K.; Houston, M.; Hanrahan, P. Brook for gpus: Stream computing on graphics hardware. *ACM Trans. Gr.* **2004**, *23*, 777–786. [CrossRef]
17. Gummaraju, J.; Morichetti, L.; Houston, M.; Sander, B.; Gaster, B.R.; Zheng, B. Twin peaks: A software platform for heterogeneous computing on general-purpose and graphics processors. In Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, Vienna, Austria, 11–15 September 2010; pp. 205–216.
18. Lee, V.W.; Kim, C.; Chhugani, J.; Deisher, M.; Kim, D.; Nguyen, A.D.; Satish, N.; Smelyanskiy, M.; Chennupaty, S.; Hammarlund, P.; et al. Debunking the 100x GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU. *ACM Sigarch Comput. Archit. News* **2010**, *38*, 451–460. [CrossRef]
19. Shen, J.; Varbanescu, A.L.; Sips, H.; Arntzen, M.; Simons, D.G. Glinda: A framework for accelerating imbalanced applications on heterogeneous platforms. In Proceedings of the ACM International Conference on Computing Frontiers, Ischia, Italy, 14–16 May 2013; pp. 1–10.
20. Jang, H.; Park, A.; Jung, K. Neural network implementation using cuda and openmp. In Proceedings of the Digital Image Computing: Techniques and Applications (DICTA), Canberra, Australia, 1–3 December 2008; pp. 155–161.
21. Huang, F.; Bu, S.; Tao, J.; Tan, X. Opencl implementation of a parallel universal kriging algorithm for massive spatial data interpolation on heterogeneous systems. *ISPRS Int. J. Geo. Inf.* **2016**, *5*, 96. [CrossRef]
22. Shi, X.; Ye, F. Kriging interpolation over heterogeneous computer architectures and systems. *GISci. Remote Sens.* **2013**, *50*, 196–211.
23. Yan, C.; Liu, J.; Zhao, G.; Chen, C.; Yue, T. A high accuracy surface modeling method based on GPU accelerated multi-grid method. *Trans. GIS* **2016**, *20*, 991–1003. [CrossRef]
24. Belviranli, M.E.; Bhuyan, L.N.; Gupta, R. A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Trans. Archit. Code Optim.* **2013**, *9*, 1–20. [CrossRef]
25. Brodtkorb, A.R.; Dyken, C.; Hagen, T.R.; Hjelmervik, J.M.; Storaasli, O.O. State-of-the-art in heterogeneous computing. *Sci. Prog.* **2010**, *18*, 1–33. [CrossRef]
26. Duchon, J. *Splines Minimizing Rotation-Invariant Seminorms in Sobolev Spaces*; Springer: Berlin/Heidelberg, Germany, 1977.
27. Powell, M.J.D. *A Review of Algorithms for Thin Plate Spline Interpolation in Two Dimensions*; World Scientific Publishing: London, UK, 1996; pp. 303–322.
28. Mitas, L.; Mitasova, H. Spatial interpolation. In *Geographical Information Systems: Principles, Techniques, Management and Applications*; Wiley: New York, NY, USA, 1999; pp. 481–492.