Journal of Zhejiang University-SCIENCE C (Computers & Electronics)
ISSN 1869-1951 (Print); ISSN 1869-196X (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

Accelerating geospatial analysis on GPUs using CUDA*

Ying-jie XIA[†1,2,3], Li KUANG[1], Xiu-mei LI[1]

([1]*Hangzhou Institute of Service Engineering, Hangzhou Normal University, Hangzhou 310012, China*)

([2]*Department of Automation, School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai 200240, China*)

([3]*Provincial Key Laboratory for Computer Information Processing Technology, Soochow University, Soochow 215006, China*)

[†]E-mail: xiayingjie@zju.edu.cn

Received Mar. 2, 2011; Revision accepted June 30, 2011; Crosschecked Nov. 4, 2011

Abstract: Inverse distance weighting (IDW) interpolation and viewshed are two popular algorithms for geospatial analysis. IDW interpolation assigns geographical values to unknown spatial points using values from a usually scattered set of known points, and viewshed identifies the cells in a spatial raster that can be seen by observers. Although the implementations of both algorithms are available for different scales of input data, the computation for a large-scale domain requires a mass amount of cycles, which limits their usage. Due to the growing popularity of the graphics processing unit (GPU) for general purpose applications, we aim to accelerate geospatial analysis via a GPU based parallel computing approach. In this paper, we propose a generic methodological framework for geospatial analysis based on GPU and its programming model Compute Unified Device Architecture (CUDA), and explore how to map the inherent parallelism degrees of IDW interpolation and viewshed to the framework, which gives rise to a high computational throughput. The CUDA-based implementations of IDW interpolation and viewshed indicate that the architecture of GPU is suitable for parallelizing the algorithms of geospatial analysis. Experimental results show that the CUDA-based implementations running on GPU can lead to dataset dependent speedups in the range of 13–33-fold for IDW interpolation and 28–925-fold for viewshed analysis. Their computation time can be reduced by an order of magnitude compared to classical sequential versions, without losing the accuracy of interpolation and visibility judgment.

Key words: General purpose GPU, CUDA, Geospatial analysis, Parallelization
doi:10.1631/jzus.C1100051 **Document code:** A **CLC number:** TP391

1 Introduction

Graphics processing unit (GPU), traditionally known as the video card, is used for rendering graphical information onto a screen. Recently, the emergence of general purpose GPU (GPGPU) breaks this obsolete view by evolving into a multithreaded and multicore processor with tremendous computational power and high memory bandwidth. Programmability of GPUs was originally limited by application programming interfaces (APIs) such as OpenGL that was designed for graphical workloads. By changing to the parallel architecture of GPGPUs, new programming frameworks, such as NVIDIA's Compute Unified Device Architecture (CUDA) (NVIDIA, 2010) and AMD's OpenCL (AMD, 2010), have eased the difficulty of programming on highly parallel GPU platforms. The process of developing efficient GPU implementations is highly application-dependent. For example, the data parallel applications, in which different data are processed in parallel using the same code, can greatly benefit from their adaption to GPU implementations.

Geospatial analysis is a term widely used to describe the combination of spatial software and analytical methods with terrestrial or geographic datasets. It is often used in conjunction with a geographic information system (GIS) and geomatics, and each GIS

* Project supported by the National Natural Science Foundation of China (No. 61002009) , the Science and Technology Planning Project of Zhejiang Province (No. 2010C31018), and the Scientific Research Fund of Hangzhou Normal University (No. HSKQ0042), China

product always employs the geospatial analysis in a specific and narrow context. One popular algorithm of geospatial analysis, inverse distance weighting (IDW) interpolation (Shepard, 1968), assigns values to unknown spatial points using values from a usually scattered set of known points. Another popular algorithm of geospatial analysis, viewshed (ESRI, 2009), identifies the cells in a spatial raster that can be seen by one or more observers. Their advantages in analyzing geospatial datasets have stimulated research into their validity, as well as on their execution performance and scalability.

However, IDW interpolation and viewshed are highly computation-intensive, especially for a large-scale space which contains a mass of points and takes an order of hours of sequential execution. There have been numerous efforts to reduce execution time for the algorithms of IDW interpolation and viewshed. Densham and Armstrong (1998) proposed several parallel algorithms for spatial analysis, including IDW interpolation. They focused mainly on the algorithms, without considering the matching between the algorithm and the computing facility. Guan and Wu (2009) used OpenMP to program IDW interpolation code into a parallel version, and run it on the multicore platform. However, because of the overhead to manage threads, this parallel implementation cannot improve the performance in a linear order. To reduce computation intensity, Papari and Petkov (2009) proposed to select the first K neighbors of a given point, instead of the entire dataset, to carry out IDW interpolation. However, by using the optimization based on the K-nearest neighbors the accuracy of the interpolation is decreased. Tournier and Naef (2010) investigated the performance of several IDW implementations on different single instruction multiple data (SIMD) architectures, especially two main classes, the architecture integrated in CPU and the architecture located on a dedicated board such as the GPGPUs. Since the loaded data points are from a 2D plane surface, they can be simply scattered equally, without considering computational intensity balance. Xia *et al*. (2010b) implemented a workflow-serialized parallel spatial IDW interpolation limited to Windows HPC. It requires a pre-processing of domain decomposition, as well as a post-processing of data gathering and a manual visualization.

As for viewshed, Ware *et al*. (1998) employed some domain partitioning strategies to implement parallel visibility analysis on a cluster of workstations. Kidner *et al*. (2002) designed a multiscale implicit triangulated irregular network (TIN) for the intervisibility analysis at multiple resolutions. Mineter *et al*. (2003) implemented a Complete Intervisibility Database for viewshed analysis by high-throughput computing without synchronization on multiple distributed processors. These viewshed related works are all concerned with their parallel implementation on distributed-memory systems, in which there is communication between processes across a network. Xia *et al*. (2010a) implemented parallel viewshed on GPU; however, it is still a specialized case and cannot be generalized to a framework for accelerating geospatial analysis using CUDA.

However, to the best of our knowledge, there is little work on how the algorithms of geospatial analysis can be mapped to GPU architecture. The computation, memory, and communication possibilities of this new multicore hardware can provide a convenient platform for the execution of algorithms that are partly or completely parallel. In this paper, we focus on the research of accelerating geospatial analysis on GPU, and implement CUDA-based IDW interpolation and viewshed packages by mapping the algorithms on the generic framework. This work shows that the resources and degrees of parallelism provided by GPU can be conveniently exploited to analyze a large amount of spatial data. To our knowledge, the implementations of CUDA-based IDW interpolation and viewshed are pioneering through their use of GPGPUs for spatial analysis.

2 Algorithms of geospatial analysis

In this section we briefly review two representative algorithms of geospatial analysis, IDW interpolation and viewshed. They can be applied to data-intensive computations of spatial data analysis.

2.1 IDW interpolation

IDW interpolation, or inverse distance weighting interpolation, is one of the most popular multivariate interpolation algorithms in the geographic information system (GIS). IDW interpolation is widely used for spatial interpolation to produce approximately continuous surfaces from discrete data points. The

process to produce such surfaces involves considerable computation, especially when large-scale problems are addressed.

In the proposed IDW interpolation implementation, a quadtree-based domain decomposition algorithm is first applied to decompose the input spatial data for load balancing. The quadtree is a tree data structure in which each internal node has exactly four children (Finkel and Bentley, 1974). It is widely used to partition a 2D space by recursively subdividing it into four quads in arbitrary shape. Fig. 1 shows an example of the quadtree-based domain decomposition algorithm.

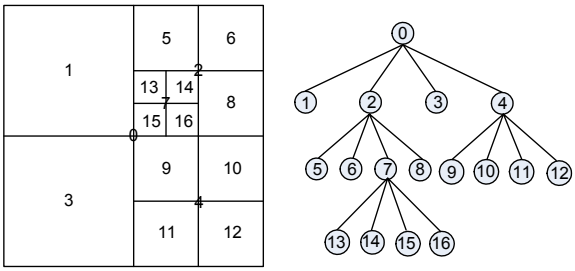

Fig. 1 An example of the quadtree-based domain decomposition algorithm

The domain can be decomposed into quads of different sizes according to the decomposition levels. Note that the decomposition levels are equivalent to the heights in the quadtree. Then we execute IDW interpolation in batch mode on these quads. The Clarke algorithm (Clarke, 1995) is designed specifically to reduce the amount of searching required to compute interpolated *Z*-values. The *Z*-value of any desired point p is computed using the following equation:

$$Z_p = \left(\sum_{i=1}^{k} z_i / d_i^{\beta}\right) \Big/ \left(\sum_{i=1}^{k} 1/d_i^{\beta}\right), \qquad (1)$$

where Z_p is the interpolated value at point p, z_i is the observed value at control point i in the neighborhood of p, k is the number of points used in the interpolation in the neighborhood of p, d_i is the Euclidian distance from i to p, and β is the distance weighting factor.

Afterwards, the output data will be gathered and visualized using the post-processing functions. When the problem grows computation-intensive, the techniques for parallel computing can be used to improve the efficiency by an order of magnitude. This will be discussed in detail in Section 5.1.

2.2 Viewshed

Viewshed analysis is a GIS function which uses the elevation value of each cell in the digital elevation model (DEM) to determine its visibility from particular observers. It is commonly used by landscape architects or urban planners to locate communication towers, to estimate the impact of a building, etc. When the viewshed algorithm is applied into DEM in a large scale or higher resolution, it requires an appropriate parallel implementation for time-tolerance and efficiency.

As illustrated in Fig. 2, the algorithm in viewshed implementation is divided into two steps, which are finding all the rays from the observer to represent lines of sight, and doing visibility analysis for all the cells passed by each ray.

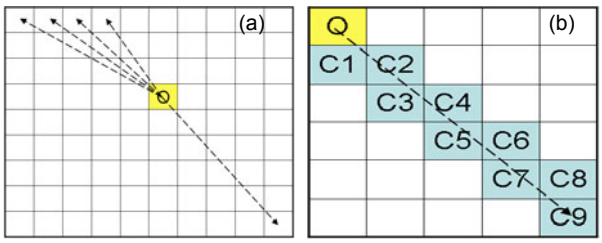

Fig. 2 Two steps of viewshed analysis
(a) Finding all the rays from the observer; (b) Doing visibility analysis for all the cells passed by each ray

In Fig. 2a, the raster represents the input DEM, which can be regarded as an elevation matrix. A ray of sight is produced from the observer identified by 'O' to another cell. As in the preliminary work, by traversing the cells of raster, we can compute all the rays. In Fig. 2b, the cells passed by one of the rays are identified by C1 to C9. To determine the visibility of a cell, we need to compare the angle of each cell between the observer and the checked cell with that of the checked cell. If the angle of any cell between the observer and the checked cell is greater than that of the checked cell, the checked cell is invisible, and vice versa. Note that the angle of a cell is calculated using the elevation and its distance to the observer. When one cell is passed by multiple rays, the visibility results will be accumulated by logical 'OR'.

3 General purpose GPU

The popular GPU typically handles computation for computer graphics, while the general purpose GPU is a technique to perform scientific computation in applications traditionally handled by CPU. It is made possible by the addition of programmable stages and higher precision arithmetic to the rendering pipelines, which allows developers to use stream processing on non-graphics data (Nickolls *et al*., 2008). Since NVIDIA released CUDA in 2007, a variety of parallel programs have been developed to run on GPUs for different applications, including molecular mechanics (Pratas *et al*., 2010), computational fluid dynamics (Micikevicius, 2009), and computer graphics (Zhou *et al*., 2008). Nevertheless, GPUs cannot speed up all possible applications. The algorithms implemented by CUDA need to explicitly express parallelism through the execution of thousands of threads, to make available resources in the GPU be efficiently occupied (NVIDIA, 2010). As two main components of GPGPU, the CUDA programming model and memory hierarchy are briefly introduced as follows.

3.1 CUDA programming model

The CUDA programming model is based on a logical representation of grids, blocks, and threads (NVIDIA, 2010). It is called by the kernel procedures of executables and mapped to the physical representation in runtime. As shown in Fig. 3, grids are composed of blocks in two dimensions, and blocks are organized by threads in three dimensions. The number of threads that run simultaneously on a block is called warp, and the number of blocks that run simultaneously on a grid is hardware-dependent. Threads within a block can cooperate amongst themselves by sharing data through some shared memories and synchronizing their execution to coordinate memory accesses. Consequently, the programmers need to code for the data partition in order to occupy as many physical threads as possible, to achieve maximum acceleration of algorithms.

3.2 CUDA memory hierarchy

CUDA threads may access data from a hierarchy of memories during their execution. As illustrated in Fig. 4, each thread has private local memory, and each thread block has shared memory visible to all threads of that block. All threads of the grid have access to the same global memory. The local memory, shared memory, and global memory are accessible at the thread level with read/write operations. There are two types of additional read-only memory space, constant and texture, which are accessible at the grid level.

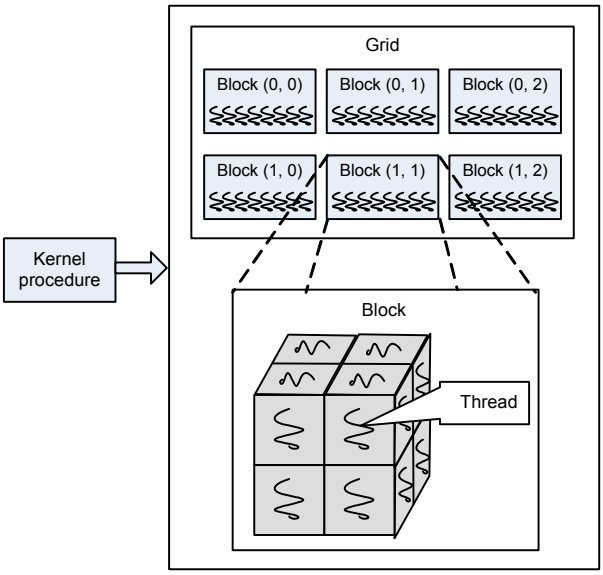

Fig. 3 Logical representation of the CUDA programming model

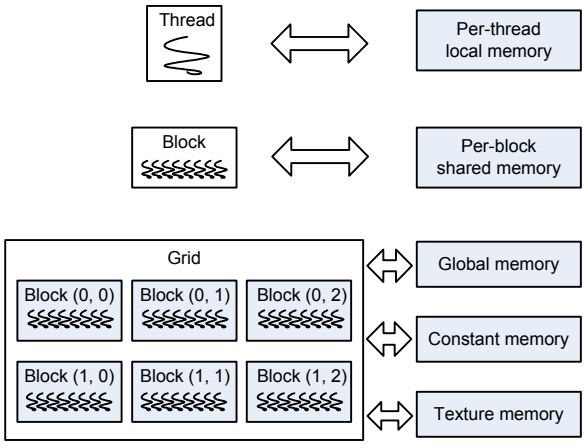

Fig. 4 CUDA memory hierarchy

4 Methodological framework for geospatial analysis

As introduced in Section 2, the two algorithms, IDW interpolation and viewshed, focus on the data-

intensive computations of geospatial analysis. Based on detailed analyses of these two algorithms, a generic methodological framework consisting of domain decomposition and thread allocation is developed to capture the spatial structural characteristics, which can be generalized for implementing geospatial analysis (Wang and Armstrong, 2009).

4.1 Domain decomposition

In a variety of fields such as image processing, GIS, and remote sensing (Gaede and Guenther, 1998; Samet, 2006), methods have been employed to adaptively decompose two-dimensional spatial data. No matter which method of domain decomposition is adopted, the aim is to generate subdomains with relative independence between each other. The proposed methodological framework generalizes different methods, such as region quadtrees and line of sight, to capture spatial heterogeneity and distribution characteristics for decomposing the spatial computational domain. The subdomains can be defined in either partitioning or overlap mode. Moreover, the subdomains are represented as instances of specific data structure, which are linked and stored in a single directional list. Eventually, for optimization, one or more instances are grouped linearly to represent an individual task which is scheduled to the available computing resource. This linear combination provides a means to control the size of an individual task, which is equivalent to its computation workload.

To combine these instances adaptive to spatial dependency in the spatial computational domain, the challenge is that there exists no total ordering among subdomains to preserve their spatial proximity. The common strategy in our framework is first to decompose the spatial domain in various shapes, and then label each cell with a unique number to define its position in an ordering for thread allocation and computation.

4.2 Thread allocation

The CUDA programming model has been illustrated in Section 3. The subdomains from domain decomposition are allocated with GPU physical threads. In general, since the dimensionality of the geospatial subdomain is smaller than or equal to three, it can match the architecture well. The grid and block of threads are logically organized as quad or line, up to specific shapes of subdomains. The methodological framework exploits the proximity preserving property to produce different sizes of spatial computational subdomains that are adaptive to spatial dependency, distribution, and heterogeneity in the underlying spatial domains. Therefore, different numbers of threads are allocated to different subdomains. The number of allocated threads matches the size of the subdomain. In the framework, if the number of threads allocated is smaller than the subdomain size, the allocation should be repeated sequentially to cover the whole computational domain.

4.3 Granularity

The size of the decomposed subdomain represents the granularity of the spatial computational domain representation. This granularity, chosen to control the computational intensity of each subdomain, is determined by two competing principles: (1) The granularity must be sufficiently coarse to ensure that the domain decomposition is computationally inexpensive; (2) The granularity must be sufficiently fine to produce a large number of subdomains that can be executed in maximally parallel. It is a trade-off problem to determine the optimal granularity for a spatial computational domain representation based on these two principles, and the feasible solution is to achieve homogeneity of computational intensity within any individual spatial computational subdomain representation. Homogeneity can be achieved during domain decomposition, which is investigated based on pattern recognition and matching approaches, such as feature tracking and object recognition.

4.4 Generality of the framework

This methodological framework is generic, and can be applied to other geospatial analyses besides the two algorithms demonstrated in Section 2. Generally speaking, for a given analysis, the framework is theoretically composed of domain decomposition and thread allocation. A detailed analysis of each algorithm is required, however, to specify spatial domain decomposition that yields subdomain for each GPU threads unit. In other cases, if the primary contributing sources of computational intensity are well known for a particular type of analysis, the subdomains that correspond to the primary sources need to be regarded in thread allocation.

5 CUDA-based implementation of geospatial analysis

Both IDW interpolation and viewshed are implemented by breaking down the task into subtasks with balanced computation intensity. These subtasks are assigned to run on the threads of GPU through CUDA. To achieve the optimal mapping between the logical programs and physical resources, the number of subtasks and their computation intensities should be evaluated, and the algorithms should be developed based on the generic methodological framework. We also need to determine which part of the logical program is coded to be sequentially executed on CPU and which part of program is coded based on CUDA and executed on GPU in parallel and cooperatively.

5.1 CUDA-based IDW interpolation

The Clarke algorithm, which is a parallel implementation of the complete IDW interpolation, consists mainly of four steps, that is, spatial domain decomposition, interpolation, output data gathering, and visualization. Since its basic elements include a *K*-nearest neighbor searching for each point, the Clarke algorithm is computation-intensive. Therefore, spatial domain decomposition is introduced to parallelize the interpolation to minimize the computation workload. In the implementation, a Morton ordered quadtree is constructed to decompose the spatial domain and produce scalable interpolation subtasks sensitive to the input dataset. The output data are separated into different DEM files corresponding to the decomposed quads, and therefore all the DEM files need to be gathered like a jigsaw puzzle and rendered for visualization. The spatial domain decomposition, output data gathering, and visualization are not computation-intensive and will not improve much efficiency by adapting into parallelized versions. Therefore, in this study we focus on the CUDA-based implementation of IDW interpolation.

Generated from domain decomposition based on the quadtree algorithm, the quads are leveraged following the rule of balancing interpolation workloads allocated to the threads of GPU. Each quad is assigned to a block of threads whose dimensionality is defined to match the maximum size of quad. In Fig. 5, the spatial domain with one cluster of points is decomposed by the quadtree algorithm to decomposition level three. The dense subdomains map to the quads of smaller size due to their high computation workloads. As illustrated in Fig. 5, either the larger quads of 8×8 points or smaller quads of 4×4 points are aligned to the GPU block size of 8×8. Each thread calculates the interpolation value of one point using IDW interpolation on the limited input points in its neighborhood.

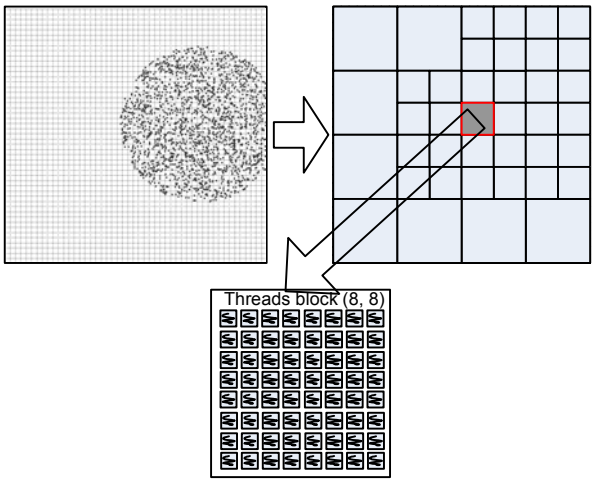

Fig. 5 Spatial domain decomposition on input data with one cluster of points and the quads assignment to the GPU threads blocks of the maximum size

The CUDA-based IDW interpolation program is designed and coded to execute domain decomposition, data gathering, visualization on CPU, and interpolation on GPU, respectively. This implementation accelerates geospatial analysis by an order of magnitude, especially for vast spatial domains.

5.2 CUDA-based viewshed

The viewshed is separated into two steps, implemented as two layered components, matrix traversal and ray traversal. Either of these two traversals can be coded in parallel by CUDA or in sequential iteration. Therefore, as shown in Table 1, there are four combinations for the viewshed implementation, that is, sequential matrix traversal and sequential ray traversal (SMSR), parallel matrix traversal and sequential ray traversal (PMSR), sequential matrix traversal and parallel ray traversal (SMPR), and parallel matrix traversal and parallel ray traversal (PMPR). For all the combinations, we must ascertain how many rays are produced in matrix traversal and how many cells are passed by each ray in ray traversal.

In matrix traversal, we traverse all the cells, instead of only the boundary cells proposed by Blelloch (1990), to produce rays. In ray traversal, we adopt the hop method with a specified step size to determine the cells passed by each ray. Interfaces are left for both solutions for easy modifications. Since SMSR uses sequential iterations to traverse all cells of raster to create rays and traverse all cells passed by each ray, it will be regarded as the comparison case in the experiments. Here we concentrate the analysis on CUDA-based viewshed implementation of PMSR, SMPR, and PMPR.

Table 1 Four combinations for viewshed implementation

Viewshed implementation	Matrix traversal	Ray traversal
SMSR	Sequential	Sequential
PMSR	Parallel	Sequential
SMPR	Sequential	Parallel
PMPR	Parallel	Parallel

1. PMSR

The matrix traversal of PMSR is implemented in parallel on GPU, by allocating one physical thread to produce one ray between the observer and any cell of raster except the observer itself, while the ray traversal of PMSR is executed sequentially by CPU. Each GPU physical thread iteratively traverses all the cells passed by one ray from the nearest to the furthest to the observer. If the elevation angle of the line linked between the observer and the checked cell is greater than the maximum angle of cells nearer to the observer, its checked visibility is set true and its elevation angle replaces the original maximum angle, and vice versa. The pseudo-code of PMSR is as follows.

```
PMSR
// Parallel, traverse all the cells except the observer itself
Each thread in GPU creates one ray between observer and one cell in matrix
  // Iteration, traverse from the nearest to the furthest to the observer
  For each cell passed by one ray {
    // Get the elevation angle of the line between observer and cell
    getAngle(cell);
    If (this angle>the maximum angle of the cells nearer to observer) {
      visibility[cell]=true;
      maximum angle=this angle;
    }
  }
                                        Run in one GPU thread
```

2. SMPR

The matrix traversal of SMPR is executed sequentially by CPU to produce all the rays from the observer, while the ray traversal runs in parallel on GPU. The threads of GPU are first allocated to calculate the angles of all cells passed by each ray, and then use CUDA library function cudppScan to perform a max-scan operation on the calculated angles of the cells passed by one ray, and put the maximum angles ever scanned into an array. Afterwards, each physical thread of GPU compares the angle of each cell in one ray with the corresponding max-scanned angle in the same position of array. If the angle of the checked cell is equal to the corresponding max-scanned angle, the cell is visible, and vice versa. These three operations, angle calculation, max-scan, and visibility judgment, are all implemented as CUDA kernel procedures, which are parallelized on GPU. The pseudo-code of SMPR is as follows.

```
SMPR
// Iteration, traverse all the cells except the observer itself
For each ray created between observer and each cell in matrix {
  // Parallel
  Calculate the angles of all the cells passed by one ray;

  // Parallel, scan from the nearest to the furthest to the observer
  // Put the maximum angle ever scanned to each element of the array
  Perform a max-scan on the calculated angles of the cells passed by one ray;

  // Parallel, if angle_array[i]==max-scan_array[i], the cell is visible
  Compare each in the same position of angle array and its max-scan array;
                                        Three CUDA kernel procedures run on threads of GPU
}
```

3. PMPR

PMPR is a complete parallel implementation of viewshed. As depicted in Fig. 6, PMPR rolls out the work of both matrix traversal and ray traversal on threads matrix of GPU.

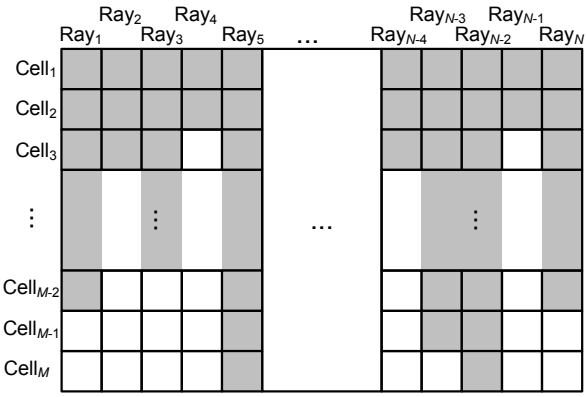

Fig. 6 Threads matrix allocation on GPU for PMPR
The grey elements represent the physical threads allocated for visibility calculation of all the cells on all the rays

However, this is an ideal mode because in practical applications the number of physical threads manufactured per GPU is limited. If a large scale of raster requires an overwhelming number of threads for parallelized calculations, PMPR may not be an efficient implementation. The pseudo-code of PMPR is as follows.

```
PMPR
  // Parallel, traverse all the cells passed by all the rays
  Calculate the angles of all the cells passed by all the rays;

  // Parallel, traverse all the rays
  // For each ray, scan from the nearest to the furthest to observer
  // For each ray, put the maximum angle ever scanned to an array
  For all the rays, perform a max-scan on the calculated angles of the cells
  passed by each ray;

  // Parallel, traverse all the rays
  // For each ray, if angle_array[i]==max-scan_array[i], the cell is visible
  For all the rays, compare each element in the same position of angle array
  and its max-scan array of each ray;

                    Three CUDA kernel procedures run on threads of GPU
```

To be different from PMSR and SMPR, PMPR implements all the angle calculation, max-scan, and visibility judgment by CUDA kernel procedures, which are applied to all the cells passed by all the rays. Therefore, these three operations can be run in parallel on GPU.

6 Experimental results and analysis

In this section, we report our experimental results and analysis, focusing on the scalability of our GPU implementation compared to a CPU baseline. For IDW interpolation, we test four cases of different input datasets which are spatial domains containing zero, one, two, and six clusters of points, respectively. We also analyze the computation time for each test case executed in parallel on GPU and sequentially on CPU. The experiments for viewshed are undertaken on DEMs of different sizes with the observer setting at different positions. We run the four implementations of SMSR, PMSR, SMPR, and PMPR on all the cases. Both CUDA-based IDW interpolation and viewshed are executed on NVIDIA Quadro FX 5600 GPU, using CUDA version 2.0. The CPU host system is equipped with two dual-core 2.4 GHz AMD Opteron CPUs with 8 GB RAM, running Red Hat Enterprise Linux 5. The experimental results and analysis of IDW interpolation and viewshed are shown as the following.

6.1 Experiments on IDW interpolation

The computation time of IDW interpolation is tested. The test cases are denoted as IDW0, IDW1, IDW2, and IDW6 whose numbers stand for the cluster number of points in input datasets. For all the cases, we test both CUDA-based parallel and sequential versions to show the acceleration supported by GPU. Their comparisons are illustrated in Fig. 7.

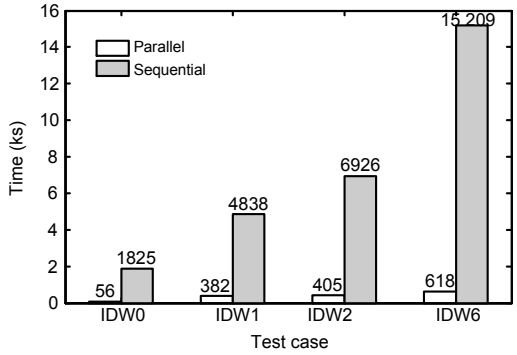

Fig. 7 Parallel and sequential computation time for the four IDW interpolation test cases

According to Fig. 7, the CUDA-based parallel implementation of IDW interpolation achieves 13–33-fold speedups in computation time over the sequential version. The speedups depend on the number of decomposed quads, which are 64, 544, 604, and 1156 respectively for our four test cases, and the computation workload of each quad. A fixed number (i.e., 64) of physical GPU threads are allocated to the quads in different cases. Since the computation of IDW0 is lightweighted and the number of threads can meet the requirements of all its quads concurrently, IDW0 achieves the maximum speedup. However, the numbers of decomposed quads in IDW1, IDW2, and IDW6 are much more than 64. When the domain decomposition is finer grained, the computation workload for each quad decreases. Therefore, their parallel computation times increase with their numbers of quads, while their speedups also increase from IDW1 to IDW6 because of the decreasing main workload for quad processing.

The result data of parallel interpolation are gathered and rendered into images. All the visualized images are shown in Fig. 8. Each image contains the corresponding cluster number of spatial points, which can verify the exactness of parallel implementation.

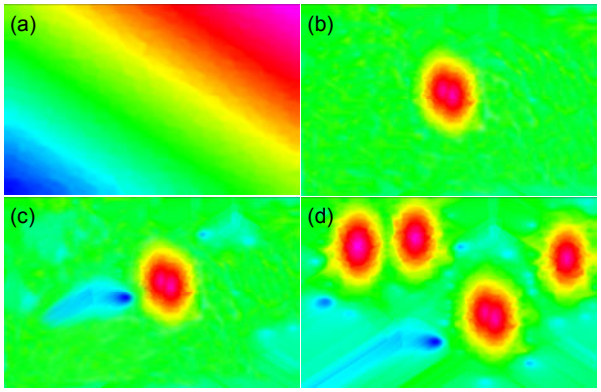

Fig. 8 Visualizing results of parallel IDW interpolation for IDW0 (a), IDW1 (b), IDW2 (c), and IDW6 (d)

6.2 Experiments on viewshed

To test the scalability of viewshed computation as the size of input increases by an order of magnitude, we use the test cases including DEM1 of 50 columns by 30 rows, DEM2 of 600 columns by 400 rows, and DEM3 of 4996 columns by 3088 rows. Since the position of the observer affects the creation of rays thereby influencing the number of cells passed, we set the observer at the origin (top left corner) and center of DEMs. Table 2 shows the coordinates of the observers for different test cases.

Table 2 Coordinates of the observer for different DEMs

Test case	Size	Observer origin	Observer center
DEM1	30×50	(0, 0)	(25, 15)
DEM2	400×600	(0, 0)	(300, 200)
DEM3	3088×4996	(0, 0)	(2498, 1544)

We test SMSR, PMSR, SMPR, and PMPR with the observer at the origin and the center; therefore, each DEM has eight test cases. The experimental results are listed in Table 3. The computation time of PMSR is several orders of magnitude less than those of the other three implementations, measured as approximately 28–925-fold speedups. This is because PMSR achieves a better trade-off between the acceleration of parallel matrix traversal by CUDA and the workloads brought for data distribution and code copy. In contrast, for SMPR, the matrix traversal is executed in sequential iteration on CPU, and the ray traversal to determine the visibilities of cells is implemented in parallel on GPU. Compared with PMSR, the implementation of SMPR runs the outer traversal in sequential iteration which sets the keynote to determine the efficiency of the whole implementation. Since the ray traversal is not a data-centric and computation-intensive part, it cannot be well accelerated by CUDA. Moreover, in SMPR the outer iterative traversal and inner parallel traversal are interleaved; therefore, its switch and memory copy between CPU and GPU cost many cycles. This is quite different from PMSR. Although PMSR is also a combination of parallel and sequential iteration, all their computations are carried out on GPU. Therefore, from the experimental results we can find that SMPR is even less efficient than the complete sequential version, SMSR. Finally, PMPR appears to be completely parallel, but too many thread allocations bring a large amount of workload on memory allocation, threads scheduling, data communications, etc., which counteract the advantages of parallelization from CUDA.

Table 3 Experimental results of four viewshed implementations

Test case	Time*		
	DEM1	DEM2	DEM3
SMSR	(4.48, 2.33) ms	(9.06, 4.94) s	(82.20, 41.47) min
PMSR	(0.16, 0.11) ms	(11.74, 5.52) ms	(5.33, 2.70) s
SMPR	(107.40, 108.00) ms	(17.80, 17.93) s	(124.32, 132.68) min
PMPR	(0.64, 0.34) ms	(9.21, 2.26) s	(65.30, 32.45) min

* In the brackets, the former number is the time for the origin, and the latter for the center

Therefore, this experiment shows that the CUDA-based implementation PMSR can achieve a better performance by good matching between a parallelized scale of problems and device configurations.

7 Conclusions and future work

The rise of GPUs as massive parallel processors opens up a wide range of opportunities for the acceleration and scaling of geospatial analysis algorithms, i.e., IDW interpolation and viewshed. IDW interpolation is adopted to produce continuous geospatial surfaces from discrete data points in GIS, and viewshed is a well established GIS research issue to determine the visibility of geospatial domains from observers. The data parallel nature of both algorithms fits the set of problems that general purpose GPUs are desired to solve. Moreover, previous research in accelerating IDW interpolation and viewshed in multi-

processor systems or scaling them by algorithms can be ported to smaller and cheaper GPU where memory systems are flexible and communications are considerably faster than networked environments. It has been shown in this paper that the implementations of IDW interpolation and viewshed based on CUDA running on GPU can lead to dataset dependent speedups in the range of 13–33-fold for IDW interpolation and 28–925-fold for viewshed analysis. These implementations reduced the computation time by more than an order of magnitude while maintaining the accuracy of geospatial analysis. In conclusion, our work showed that the GPU programming model conveniently allowed the execution of multiple subtasks in parallel over the same global memory. This fact benefits the efficiency not only because of parallel execution, but also due to the reusability of data across subtasks.

In future work, we will explore mapping additional GIS applications to the GPU, and particularly focus on how to accelerate geospatial analysis on a large scale of domain, which can be extremely computation-intensive. The algorithms for geospatial analysis can be optimized, such as implementing domain decomposition using CUDA in IDW interpolation and multiple observers support in viewshed. We would also like to consider how the algorithms will scale on AMD GPUs, and pursue further implementation using state-of-the-art Fermi-based GPU and CUDA 4.0.

References

AMD, 2010. ATI Stream Computing OpenCL Programming Guide. Available from http://developer.amd.com/gpu_assets/ATI_Stream_SDK_OpenCL_Programming_Guide.pdf [Accessed on Feb. 18, 2011].

Blelloch, G., 1990. Vector Models for Data-Parallel Computing. MIT Press, Cambridge, MA.

Clarke, K.C., 1995. Analytical and Computer Cartography (2nd Ed.). Prentice-Hall, Englewood Cliffs, NJ.

Densham, P.J., Armstrong, M.P., 1998. Spatial Analysis. *In*: Healey, R.G., Dowers, S., Gittings, B.M., *et al.* (Eds.), Parallel Processing Algorithms for GIS. Taylor and Francis, London, p.387-413.

ESRI, 2009. ArcGIS Desktop Help. Available from http://webhelp.esri.com/arcgisdesktop/9.3/index.cfmTopicName=Performing_a_viewshed_analysis [Accessed on Feb. 18, 2011].

Finkel, R., Bentley, J.L., 1974. Quad Trees: a data structure for retrieval on composite keys. *Acta Inform.*, **4**(1):1-9. [doi:10.1007/BF00288933]

Gaede, V., Guenther, O., 1998. Multidimensional access methods. *ACM Comput. Surv.*, **30**(2):170-231. [doi:10.1145/280277.280279]

Guan, X., Wu, H., 2009. Parallel optimization of IDW interpolation algorithm on multicore platform. *SPIE*, **7146**: 71461Y-71461Y-9. [doi:10.1117/12.813163]

Kidner, D.B., Sparkes, A.J., Dorey, M.I., Mark Ware, J., Jones, C.B., 2002. Visibility analysis with the multiscale implicit TIN. *Trans. GIS*, **5**(1):19-37. [doi:10.1111/1467-9671.00065]

Micikevicius, P., 2009. 3D Finite Difference Computation on GPUs Using CUDA. Proc. 2nd Workshop on General Purpose Processing on Graphics Processing Units, p.79-84. [doi:10.1145/1513895.1513905]

Mineter, M., Dowers, S., Caldwell, D., Gittings, B., 2003. High-Throughput Computing to Enhance Intervisibility Analysis. 7th Int. Conf. on GeoComputation, p.1-10.

Nickolls, J., Buck, I., Garland, M., Skadron, K., 2008. Scalable parallel programming with CUDA. *Queue*, **6**(2):40-53. [doi:10.1145/1365490.1365500]

NVIDIA, 2010. NVIDIA CUDA C Programming Guide, Version 3.1. Available from http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf [Accessed on Feb. 18, 2011].

Papari, G., Petkov, N., 2009. Reduced Inverse Distance Weighting Interpolation for Painterly Rendering. 13th Int. Conf. on Computer Analysis of Images and Patterns, p.509-516. [doi:10.1007/978-3-642-03767-2_62]

Pratas, F., Mata, R., Sousa, L., 2010. Iterative Induced Dipoles Computation for Molecular Mechanics on GPUs. Third ACM Workshop on General Purpose Processing on Graphics Processing Units, p.111-120. [doi:10.1145/1735688.1735708]

Samet, H., 2006. Foundations of Multidimensional and Metric Data Structures. Morgan-Kaufmann, San Francisco, CA.

Shepard, D., 1968. A Two-Dimensional Interpolation Function for Irregularly-Spaced Data. Proc. 23rd ACM National Conf., p.517. [doi:10.1145/800186.810616]

Tournier, J.C., Naef, M., 2010. Influences of SIMD Architectures for Scattered Data Interpolation Algorithm. 10th IEEE Int. Symp. on Performance Analysis of Systems and Software, p.109-110. [doi:10.1109/ISPASS.2010.5452056]

Wang, S., Armstrong, M.P., 2009. A theoretical approach to the use of cyberinfrastructure in geographical analysis. *Int. J. Geograph. Inform. Sci.*, **23**(2):169-193. [doi:10.1080/13658810801918509]

Ware, J.A., Kidner, D.B., Rallings, P.J., 1998. Parallel Distributed Viewshed Analysis. Proc. 6th ACM Int. Symp. on Advances in Geographic Information Systems, p.151-156. [doi:10.1145/288692.288719]

Xia, Y., Li, Y., Shi, X., 2010a. Parallel Viewshed Analysis on GPU Using CUDA. 3rd IEEE Int. Joint Conf. on Computational Sciences and Optimization, p.373-374. [doi:10.1109/CSO.2010.12]

Xia, Y., Zhu, M., Shi, X., 2010b. A workflow-serialized parallel spatial IDW interpolation on Windows HPC. *Appl. Mech. Mater.*, **20-23**:370-375. [doi:10.4028/www.scientific.net/AMM.20-23.370]

Zhou, K., Hou, Q., Wang, R., Guo, B., 2008. Real-time KD-tree construction on graphics hardware. *ACM Trans. Graph.*, **27**(5):126. [doi:10.1145/1409060.1409079]