A Parallel Computing Approach to Viewshed Analysis of Large Terrain Data Using Graphics Processing Units

Abstract

Viewshed analysis, often supported by Geographic Information Systems (GIS), is widely used in many application domains. However, as terrain data continue to become increasingly large and available at high resolutions, data-intensive viewshed analysis poses significant computational challenges. General-Purpose computation on Graphics Processing Units (GPGPU) provides a promising means to address such challenges. This paper describes a parallel computing approach to data-intensive viewshed analysis of large terrain data using GPUs. Our approach exploits high-bandwidth memory of GPUs and parallelism of massive spatial data to enable memory-intensive and compute-intensive tasks while CPUs (Central Processing Units) are used to achieve efficient Input/Output (I/O) management. Furthermore, a two-level spatial domain decomposition strategy is developed to mitigate a performance bottleneck caused by data transfer in the memory hierarchy of GPU-based architecture. Computational experiments were designed to evaluate computational performance of the approach. The experiments demonstrate significant performance improvement over a well-known sequential computing method, and enhanced ability of analyzing sizable datasets that the sequential computing method cannot handle.

Keywords: Viewshed analysis; General-Purpose computation on Graphics Processing Units; Parallel computing; Spatial data analysis
A Parallel Computing Approach to Viewshed Analysis of Large Terrain Data Using Graphics Processing Units

1 Introduction

Viewshed analysis is a widely used spatial analysis, often supported by Geographic Information Systems (GIS) to determine regions of a terrain that are visible from a set of specified locations (Fisher 1993). Many spatial analyses, such as site selection, path planning, navigation, landscape planning and environmental change detection, increasingly require data-intensive computation of viewshed analysis of high-resolution digital elevation datasets (Nagy 1994; Franklin and Ray 1994; De Floriani et al. 1994; Fisher 1995; De Floriani and Magillo 2003; Jiang et al. 2011). For example, an elevation dataset derived from Light Detection And Ranging (LIDAR) instruments with one sample (4 bytes used to represent each sample) per 1-meter² had the size of approximately 689 GB for the State of Washington in the U.S. (Fishman et al. 2009). To analyze spatial data of such a size or even larger, conventional sequential computing approaches become unsuitable, and the use of parallel computing – now a mainstream of scientific computing – is necessary.

When massive spatial data are treated, only a small fraction of input data can be held in the main memory and various levels of caches in a computer at any given time. Hence, data need to be transferred across the various levels of storage and memory hierarchy (e.g., disk, memory, and cache), which may contribute a major performance bottleneck to applications. Specifically for viewshed analysis, the basic computation procedure involves the calculation of visibility of a target location from a specific source.
viewpoint $v$. This calculation is usually implemented by emanating a line-of-sight (LOS) ray from viewpoint $v$ to that target location. This procedure, however, may result in irregular accesses to various parts of input terrain data, causing significant cache misses and extensive Input/Output (I/O) accesses if conducted without optimization or approximation. Therefore, the computation tends to be memory bandwidth-bound and I/O intensive.

I/O-efficient algorithms, also known as external-memory algorithms, can be developed for viewshed analysis to reduce data swapping between disk and main memory of a computer for improving the overall performance of handling large datasets (Magalhaes et al. 2007; Haverkort et al. 2009; Fishman et al. 2009). However, even with these improvements, the computing time of exact algorithms remains significant, primarily because such algorithms were predominantly designed to work on a single-processor computer system. The computation of viewshed analysis against large terrain data entails a massive number of LOS-based calculations, for which conventional sequential computing approaches are unable to support efficiently or effectively.

Parallel computing has become the mainstream of computer architecture with tremendous potential in advancing geographic information science (Wang and Armstrong, 2009; Wang and Liu, 2009). As for viewshed analysis, parallel processing techniques have been applied to improve computational performance (Mills et al. 1992; Teng et al. 1993; Strnad 2011; Fang et al. 2011; Gao et al. 2011). However, these previous work assumed that input data as a whole can be handled within the main memory of a computer. Hence, this assumption is not applicable to processing massive terrain data due to infeasibility of holding all input data within main memory during the computation of
an entire analysis. Distributed computing of viewshed analysis was also introduced in literature (De Floriani et al. 1994, Ware et al. 1998), which used a cluster of computers with each one of them processing a big chunk of decomposed input data based on sequential algorithms. These distributed computing approaches can meet the memory requirement of large input datasets. The performance improvement is, however, rather limited since each computer still uses the sequential algorithms for data processing.

This research focuses on resolving data-intensive challenges in viewshed computation through the General-Purpose computation on Graphics Processing Units (GPGPU). GPUs have been recently used in a large number of application domains, since they can provide significant computational power with affordable cost, and their programmability has improved as well (Owens et al. 2007). Generally, GPUs are designed to exploit data parallelism, and it has been reported that GPUs can achieve 10 times more floating-point operations per second (FLOPS) compared to CPUs (Govindaraju et al. 2006, Owens et al. 2007, Ryoo et al. 2008, Che et al. 2008, Kirk and Hwu 2010). Furthermore, data-intensive applications can benefit from high-bandwidth memory of GPUs (Kirk and Hwu 2010). Another advantage of using GPUs is that while GPUs are dedicated to computing tasks, CPUs can be engaged to perform I/O management functions thereby enhancing I/O performance.

The primary purpose of our research is to exploit spatial data parallelism, efficient GPU memory and I/O access in viewshed analysis of large spatial data. A parallel computing approach, based on GPU using the NVIDIA Compute Unified Device Architecture (CUDA), is designed with a focus placed on mitigating the performance bottleneck caused by data transfer across disk, CPU memory, GPU memory and
processors. Our specific strategies include: (1) a two-level spatial domain decomposition that is mapped to the memory hierarchy of a computer system (i.e., disk to CPU main memory to GPU off-chip memory, and GPU off-chip memory to on-chip fast memory); (2) reuse of data transferred from disk; and (3) a massively parallel processing method. These strategies are designed to be portable to any computer system equipped with CUDA-enabled GPUs, either personal computers or high performance cluster computers. Computational experiments have shown more than 50 times of speedup obtained without losing accuracy, by comparing our approach to a well-known sequential algorithm – \textit{R3} – (Shapira 1990) on small datasets (<4GB), as well as the capability to process large datasets that the sequential algorithm cannot handle. Moreover, by efficiently loading data into on-chip shared memory, up to 3 times of speedup were achieved compared to using off-chip memory only.

In the remainder of this paper, section 2 reviews the background related to this research. Section 3 illustrates the well-known single-processor sequential algorithm for viewshed analysis. Regarding our parallel computing approach based on GPUs, section 4 describes our specific strategies, implementations and theoretical analysis. Section 5 details the computational experiments that are designed to evaluate the performance of the GPU-based approach. The concluding section summarizes the findings of this research and future research directions.

2 Background

2.1 Visibility
Viewshed analysis of terrain can be based on either of two classes of Digital Elevation Models (DEM)s studied in geographic information science: Triangulated Irregular Networks (TINs) and Regular Square Grids (RSGs) (De Floriani and Magillo 2003). A terrain represented based on TIN consists of a collection of planar, irregular triangular patches. RSG is commonly used for terrain representation, and often consists of equally sized rectangles or grid cells. Results of viewshed analysis based on RSG are usually encoded by marking every grid cell as visible or invisible with Boolean values. Such discrete viewshed is sufficiently accurate, since grid-based terrain representation often has fine granularity (De Floriani and Magillo 2003). Furthermore, grid regularity is suitable for supporting the development of parallel processing techniques. Consequently, this research focuses on computing viewshed against RSG-based terrain datasets. The basic step of viewshed analysis of grid terrain involves the calculation of visibility of every grid cell from a given viewpoint \( v \). We will give the definition of visibility on grid terrain and introduce an algorithm we adopt for its calculation.

2.1.1 Problem definition

Let \( T \) be a grid terrain represented as a uniform grid of \( n \times n \) square cells. Without losing generality, the elevation of each grid cell’s center is chosen to represent that of the entire cell (Van Kreveld 1996; Haverkort et al. 2009; Fishman et al. 2009). Visibility of any grid cell \( P \), called target cell, is based on the line-of-sight (LOS) emanating from any given viewpoint \( v \) to the center point \( p \) of \( P \). The elevation angle of \( P \) with respect to viewpoint \( v \) is defined as:

\[
\text{ElevAngle}(P) = \arctan((E_p - E_v) / D_{vp})
\]  

(1)
where $E_p$ and $E_v$ are the elevations of $p$ and $v$, respectively. $D_{vp}$ is the distance between $p$ and $v$ (Haverkort et al. 2009). We adopt the following definition of visibility: the target cell $P$ is visible from $v$ if and only if the elevation angle of $P$ is larger than the elevation angle of any cell intersected by the LOS from $v$ to $p$ in the projection of $T$ on a horizontal plane, called domain $D$ (see Figure 1) (Van Kreveld 1996; Haverkort et al. 2009; Fishman et al. 2009). We denote the cells intersected by the LOS from $v$ to $p$ as intersected cells.

[Figure 1 near here]

2.1.2 Amanatides and Woo’s traversal algorithm

To calculate the visibility of a target cell, the cells intersected by the projected LOS from $v$ to the center of the target cell need to be identified. Bresenham’s line algorithm (Bresenham 1965) and digital differential analyzer (DDA) (Mayorov 1964) are commonly used algorithms to compute such intersected cells. Bresenham’s line algorithm is based on integer arithmetic, considering only integral row and column coordinates of cells. Though fast computation is an advantage, it is not accurate when the endpoints of a line are not at the center of cells, due to rounding errors from converting floating-point to integer coordinates. In contrast, the DDA method employs floating-point arithmetic, which leads to more accurate computation of intersection results compared to Bresenham’s line algorithm. In this research, we adopt a variant of the DDA line algorithm – the Amanatides and Woo’s traversal algorithm (Amanatides and Woo 1987).

The Amanatides and Woo’s algorithm consists of two phases: initialization and incremental traversal (Figure 2). The integer variables $X$ and $Y$ are initialized to the
starting cell’s column and row coordinates. \( t_{Vertical} \) variable is initialized as the length of the line segment from the line’s starting point to the point where the line crosses the first cell boundary in the vertical direction. \( t_{Horizontal} \) is initialized in a similar fashion where the line crosses the first cell boundary in the horizontal direction. The variables \( t_{DeltaV} \) and \( t_{DeltaH} \) indicate how far the line must move from the current cell to crossing the next cell boundary in vertical and horizontal directions respectively. The variables \( stepX \) and \( stepY \) are initialized to either 1 or -1 indicating whether \( X \) and \( Y \) are incremented or decremented as the line crosses cell boundaries.

The incremental phase of the traversal algorithm is outlined in the basic loop in Figure 2 (B). The next intersected cell at any step is determined by comparing \( t_{Vertical} \) and \( t_{Horizontal} \) variables that are incrementally updated using \( t_{DeltaV} \) and \( t_{DeltaH} \) per iteration. The smaller of these two values indicates whether the line will cross the current cell boundary in vertical or horizontal direction and consequently the identity of the next intersected cell. The impact of this cell on the visibility of the target cell is computed by a function \( ProcessCell() \) as follows: if the elevation angle of the intersected cell is larger than that of the target cell, then the target cell is marked as invisible, and the loop terminates. Otherwise, it continues to the next iteration. As a result, when the target cell is visible, the algorithm will loop until the end of the line and thus traverse all intersected cells, which equals to the number of horizontal and vertical cell boundary crossings. Overall, this algorithm is fast since it requires only one floating-point comparison, one floating-point addition, and one integer addition per iteration.

[Figure 2 near here]
2.2 GPGPU overview

Our research uses NVIDIA CUDA-enabled GPUs. The processor layout, memory hierarchy, and CUDA thread model are illustrated using Figure 3 (NVIDIA Corporation, 2009). We highlight the following features that make them suitable for data-intensive spatial analysis applications like ours.

[Figure 3 near here]

(1) **Massive data parallelism:** CUDA-enabled GPUs are designed to exploit massive data parallelism. A large number of processor cores are organized into a set of Stream Multiprocessors (SMs). These SMs are exploited to achieve high FLOPS by employing massive threads in CUDA. CUDA programming can be deemed as a Single Program Multiple Data (SPMD) programming model. Hence, application developers specify data-parallel functions called *kernels* that are invoked by CPU called *host* to run on GPU devices across a large amount of parallel threads, each executing an instance of the kernel. These threads are organized into thread blocks and grids of thread blocks, and each thread is uniquely identified by its block and thread ID.

(2) **Dedicated memory interface:** Such GPUs also have a large pool of dedicated high-bandwidth global memory, a small amount of fast on-chip memory called shared memory and a certain number of register files. These memory resources can be flexibly assigned to threads for achieving desirable memory access performance. In particular, each thread can privately access its register files. Threads within the same thread block are allowed to access on-chip shared
memory. Threads from different thread blocks can share the global memory and can communicate via kernel-wide synchronization.

(3) **Pipeline scheduling:** The massively parallel threads are scheduled to SMs in pipeline that can mitigate global memory access latency. Due to limited hardware resources, such as the size of register files, shared memory and thread scheduling slots, a SM can hold only a certain number of thread blocks simultaneously. The ratio of active threads residing in the SM to the maximum number of threads supported on a SM of the GPU is called multiprocessor occupancy. For memory bandwidth-bound applications, increasing occupancy can help better mitigate the latency of global memory accesses (NVIDIA Corporation, 2010).

In the following sections, we describe our approach to designing and implementing a method for viewshed analysis that will exploit these features of CUDA GPUs to achieve scalable computing and desirable memory performance for analyzing sizable terrain datasets.

3 **R3 sequential algorithm**

*R3* is a well-known sequential algorithm for viewshed analysis of grid terrain (Shapira 1990; Franklin and Ray 1994; Van Kreveld 1996; Izraelevitz 2003). For example, it is used in the open source GRASS GIS software. The algorithm calculates the visibility of each target cell by running a separate LOS emanating from ν to every target cell on the grid terrain, which can be implemented using the method described in Section 2.1.2. Since the target cells can be processed independently, this algorithm is well suited to
parallel processing. Though a number of approximation algorithms have been developed in viewshed analysis (Blelloch 1990; Franklin and Ray 1994; De Floriani and Magillo 2003), they are not employed in this paper since accuracy of viewshed analysis is an important consideration for a number of applications. Furthermore, enabled by the computational capabilities offered by GPUs, we believe exact computation of viewshed for large datasets can be tractable. Van Kreveld’s radial sweep algorithm (1996) presents a fast sequential algorithm for calculating viewshed with accuracy equivalent to the R3 algorithm; however, a high degree of sequential dependencies in Van Kreveld’s algorithm makes it less suitable to exploit parallelism. R3 is hence chosen as the basis for developing our parallel approach on GPUs.

[Figure 4 near here]

3.1 Computational challenges of R3 algorithm

R3 algorithm includes four major steps (Figure 4). In step 2, the algorithm loops through every target cell in the input grid, while in step 3 all intersected cells of a target cell are traversed. This traversal can employ the Amanatides and Woo’s traversal algorithm. The R3 algorithm is hence compute-intensive, with a computational complexity of $O(n^3)$ considering an input dataset containing $n^2$ raster cells. In addition, it is observed that though target cells are processed in a row-major order, their intersected cells are accessed in a sequence defined by LOS during computation. This leads to an irregular data access pattern that is prohibitively expensive because it causes significant cache misses. Furthermore, if an input dataset is too large to be fully loaded in memory, extensive disk
I/O accesses are also expected, which may cause the degradation of computational performance.

3.2 Spatial characteristics of R3 algorithm

To improve computational performance while maintaining the accuracy of analysis results, we aim to exploit the spatial characteristics exhibited by the R3 algorithm. Specifically, these characteristics include: spatial independency, locality, and occlusion that are described as follows.

(1) **Spatial independency**: Each target cell can be processed independently. This independence indicates spatial data parallelism that may be exploited to engage many parallel threads to calculate the visibility of many target cells.

(2) **Spatial locality**: Typically a cell on a grid terrain is intersected by LOS rays to multiple target cells and could be reused. Furthermore, this intersection is more likely when LOS rays to target cells in the same neighborhood are being computed. Consequently, the computation of LOS exhibits spatial locality with respect to neighboring target cells. This spatial locality makes it possible for the neighboring target cells grouped into a larger spatial unit to facilitate data reuse, which has potential to be exploited to improve data access performance.

(3) **Occlusion**: Data needed for computing visibility of grouped target cells can be occluded. A *tile* is defined as a regular (e.g., square or rectangle shaped) spatial domain consisting of neighboring target cells, and can also be organized as a set of neighboring sub-tiles with each including a smaller number of target cells. To understand occlusion, imagine that a radial line sweeps clockwise around a
viewpoint \( v \), and that the point at which the line first intersects the tile is called \textit{start point} of the tile while the \textit{end point} is specified as the radial line last intersects the identified tile. Then, all the required data for the tile are occluded by the two lines from \( v \) to the start point and to the end point of the tile. This characteristic of occlusion is useful to identify what data may be pre-fetched for grouped neighboring target cells, and to make data reuse feasible.

Massive computation, memory and I/O accesses are three primary challenges of viewshed computation using \textit{R3} algorithm for large datasets. Our GPU-based parallel computing approach described in Section 4 is developed to tackle these challenges.

\section*{4 GPU-based parallel computing approach}

Our GPU-based parallel computing approach exploits both GPUs’ features and spatial characteristics of the \textit{R3} viewshed analysis algorithm. Four specific strategies: coarse-scale spatial domain decomposition, tile-based moving window, fine-scale spatial domain decomposition, and scalable parallel processing are developed to resolve the challenges of massive computation, memory and I/O accesses. The essence of these strategies is threefold: 1) transfer data in contiguous blocks from disk to global memory to shared memory, which is achieved by the two-level (coarse- and fine-scale) spatial domain decomposition strategies; 2) reuse the transferred data on GPU device whenever possible, which is realized by our two-level decomposition and tile-based moving window strategies; and 3) employ a large number of parallel threads to perform computationally intensive visibility computation efficiently. The first two are designed to reduce the
number of memory and I/O accesses, while the third provides the significant computing power demanded by massive computation of the viewshed analysis algorithm.

In the following subsections, we first describe details of these strategies, and then conduct a formal analysis of computing intensity.

4.1 Coarse-scale spatial domain decomposition

When terrain data are large and cannot be held in memory all at once, intersected cells have to be fetched on the fly from disk in a sequence defined by LOS during calculation for each target cell. This irregular disk access may result in extensive long-latency I/O accesses and become a major performance bottleneck. In order to resolve this problem, we decompose an input grid into tiles, each of which consists of a certain number of cells. A target tile consists of a set of neighboring target cells. Intersected tiles are those occluded by lines from viewpoint v to the start point and to the end point of the target tile (Figure 5); and they can be identified using the Amanatides and Woo’s algorithm. Based on this decomposition, we pre-fetch the required intersected tiles into GPU global memory for each target tile before computation. This achieves the desired regular data access pattern for data to be loaded consecutively from disk to memory. Furthermore, because of the characteristic of spatial locality between target cells, these cells within the same target tile can share and reuse the data pre-fetched. Thus, the number of I/O accesses to disk is reduced.

This decomposition strategy is designed to operate at coarse-scale, and aims to place any target tile and all its intersected tiles in the GPU global memory at once, which imposes a limit on tile sizes and the number of tiles that can be loaded. Consequently, we
let the upper bound of the number of required intersected tiles for a target tile be equal to
the maximum number of tiles allowed in global memory, \( Max_t \), and each tile has \( n_t \times n_t \)
cells. Then the values of \( n_t \) and \( Max_t \) can be obtained as follows.

Given an input grid with \( n \times n \) cells, without losing generality, we assume that \( n \)
can be evenly divided by \( n_t \) throughout the paper and, thus, there are \( \left( n/n_t \right) \times \left( n/n_t \right) \) tiles in
the grid. In the worst case, when the viewpoint \( v \) is in the corner tile, i.e. assume the tile’s
column and row coordinates in the grid is \((0, 0)\), and the farthest target tile from \( v \) is in
the other diagonal corner of the grid, \((n/n_t, n/n_t)\), the number of required intersected tiles
is maximal and is upper bounded by

\[
2(\frac{n}{n_t} + \frac{n}{n_t}) = 4n / n_t
\]

(2)

Where \( \left( n/n_t + n/n_t \right) \) equals to the number of horizontal and vertical tile boundary
crossings, which is also the maximum number of intersected tiles along a line from \( v \) to
the farthest target tile in the grid according to the traversal algorithm (see Section 2.1.2).
\( 2(\frac{n}{n_t} + \frac{n}{n_t}) \) denotes that the number of intersected tiles occluded by two lines from \( v \) to
the start point and the end point of the target tile is at most doubled, since there is no tile
completely occluded by the two lines. The proof is illustrated in Figure 5. As a result, the
following equation is obtained.

\[
Max_t = 4n / n_t
\]

(3)
Let the global memory capacity be $M$ bytes and 4 bytes are used to represent a cell’s elevation, then $\max_t n_t^2 \times 4 = 4n / n_t^2 \times 4 \leq M$ and

$$n_t \leq M / 16n$$

(4)

Hence, in order to process any target tile in global memory, $n_t$ can be at most $M/16n$. In our implementation, we let $n_t$ be its maximal value $M/16n$ so as to minimize the total number of iterations to process $(n/n_t)^2$ target tiles.

[Figure 5 near here]

4.2 Tile-based moving window

Each time a target tile is processed, the target tile and its intersected tiles need to be transferred to GPU global memory. This may result in large data being transferred to GPU global memory from disk, which could negatively impact computational performance. To reduce the amount of data transferred to global memory, a tile-based moving window strategy is designed to exploit the spatial locality characteristic between neighboring target tiles. The strategy increases the reuse of intersected tiles that are already in the global memory and, thus, reduces the number of I/O accesses.

As illustrated in Figure 6, when a new target tile starts to be processed, the linear indices (row-coordinate * the number of tiles in a row + column-coordinate) of its intersected tiles are calculated, and compared to a list of indices of tiles that are already in global memory. Intersected tiles required by the new target tile need to be transferred into global memory and add its index to the list only in cases they are not already in. If the
maximum number of tiles that can be held in memory is exceeded, tiles in the list that are least-frequently-used and not required by the new target tile are removed from global memory. The row-major order of processing target tiles exploits the spatial locality characteristic of the underlying algorithm, which ensures improved performance by increasing data reuse. Compared to this improvement, cost incurred by this strategy (i.e. comparing and sorting the list of indices of tiles in global memory) is oftentimes trivial, since the maximal list size $Max_t = 4n/n_t$ is small considering the tile size $n_t$ is usually large.

[Figure 6 near here]

To implement this strategy, we have designed two array data structures (see Figure 6): $List_{Indices}$ array is the list of linear indices of tiles in global memory, and $List_{Tiles}$ array stores pointers referring to memory addresses of these tiles. $List_{Tiles}[i]$ and $List_{Indices}[i]$ maintain the information of the same tile. This implementation facilitates quick memory reference to access data and thus improves performance. For example, if data in a tile are requested during computation, we first use binary search in $List_{Indices}$ to find its index and return the array entry $i$, then the data within the tile can be accessed via its memory address $List_{Tiles}[i]$ directly.

4.3 Fine-scale spatial domain decomposition

On CUDA-enabled GPUs, the most favorable regular access pattern is achieved when the same instruction for threads of consecutive IDs accesses consecutive global memory locations. In this case, hardware coalesces these consecutive accesses into a consolidated access to the memory and, thus, can reduce the number of global memory accesses and
achieve efficient use of high-bandwidth global memory (Kirk and Hwu 2010). If pre-fetched tiles in global memory are accessed without data rearrangement during computation, however, consecutive threads load their data from global memory in a sequence defined by LOS and these data are oftentimes not consecutively stored. Consequently, it may result in a large amount of irregular data accesses from long-latency global memory and, thus, becomes a performance bottleneck.

To eliminate irregular data accesses from global memory, our fine-scale decomposition strategy takes advantage of fast on-chip shared memory on GPU by emphasizing the importance of exploiting multi-scale spatial parallelism. Specifically, tiles in global memory are further decomposed into sub-tiles that are organized as target sub-tiles and intersected sub-tiles, as shown in Figure 7. One target sub-tile is processed by a thread block, which pre-fetches its required intersected sub-tiles from global memory to shared memory in the desired regular access pattern, and then accesses pre-fetched data in shared memory on the fly during computation. In this way, we achieve regular data access pattern to long-latency global memory, and shift these irregular accesses to shared memory that is much faster. As a result, overall efficient memory access is achieved. Furthermore, because of the spatial locality of the $R3$ algorithm, data in shared memory can be reused and shared by cells in the same target sub-tile, which can further reduce the number of global memory accesses.

[Figure 7 near here]

Since the fine-scale decomposition strategy maps each target sub-tile to one thread block, the size of sub-tile is determined based on the size of thread block. This
mapping is described in the immediately following Section 4.4. Thread block size is often experimentally tuned, oftentimes based on the consideration of hardware specifications such as the size of register files and shared memory. The performance tuning is discussed in Section 5.

4.4 Parallel processing

The spatial data parallelism attributed to the spatial independency characteristic of the $R^3$ algorithm is exploited for engaging parallel threads to simultaneously process target cells in a tile. Figure 8 shows an example of how target cells can be mapped to threads in a block. In our approach, fine-scale parallelism is exploited by assigning each thread to process a single target cell. Hence, the total number of threads allocated on GPUs is the same as the number of target cells in the target tile. These threads are organized into thread blocks, and each of them processes one target sub-tile. There is no cooperation between different thread blocks, which are transparently scheduled to GPU SMs in pipeline. Within a thread block, threads can be coordinated through barrier synchronization and sharing data in on-chip shared memory. However, capacity of shared memory is usually small, so it cannot hold all the intersected sub-tiles required by a target sub-tile. These intersected sub-tiles need to be dynamically loaded from global memory to shared memory through the coordination of a thread block in multiple iterations. Our implementation guarantees shared memory can contain at least one intersected sub-tile.

[Figure 8 near here][Figure 9 near here]

The working process of a thread block is as follows. After the thread block has already loaded one intersected sub-tile in shared memory, each thread in this block starts
to calculate visibility of a target cell. In doing so, a thread firstly finds the starting intersected cell from the intersected sub-tile in shared memory (Figure 9), and then traverses the next intersected cells in this sub-tile if necessary based on the traversal algorithm (see Section 2.1.2). During this process, the threads may finish at different timestamps, but they are eventually synchronized so that all the threads finish processing an intersected sub-tile in shared memory. After the processing of this intersected sub-tile is finished, the block of threads will determine whether to process the next intersected sub-tile or not. If there is at least one thread in the thread block, whose target cell is still deemed to be potentially visible, this thread needs intersected cells in the next intersected sub-tiles and all the threads in the block need to load the next intersected sub-tile from global memory into shared memory. If all cells in the target sub-tile are known as invisible, then it means that the thread block finishes all the work and no more processing is needed. Otherwise, the thread block may continue its computing process until the last intersected sub-tile is processed.

[Figure 10 near here]

In our implementation, in order to control the synchronization and cooperation, we define a shared Boolean variable Flag. Every time after loading one intersected sub-tile into shared memory, it is reset as 0. After processing this sub-tile, the value of this flag could be either unchanged or updated to 1; 1 indicates that there is at least one cell in the target sub-tile being considered to be possibly visible and all the threads need to load the next intersected sub-tile, while a value of 0 means all cells in the target sub-tile are known as invisible and the thread block finishes all the work. Figure 10 outlines the pseudo code of the computing process. Overheads incurred in the computing process,
including instructions for computing intersected sub-tile indices and synchronization of threads, are insignificant compared to performance gains contributed by our parallel computing strategies applied. The performance improvements are described in Section 5.

4.5 Computing intensity

The computing intensity of our approach varies as each different viewpoint \( v \) is usually associated with a different set of visible cells \( n_v \) and invisible cells \( n_{iv} \). For each visible cell \( P_v \), our approach traverses all the intersected cells from \( v \) to \( P_v \) and, thus, the time of deriving each visible cell increases as the distance between \( v \) and \( P_v \) increases, and is denoted \( D_{vP_v} \). In contrast, for each invisible cell \( P_{iv} \), the traversal terminates once the first obstruction cell \( P_{fo} \) is found and, hence, invisible cells are potentially computed faster than visible cells. The time of deriving each invisible cell also increases with the distance between \( v \) and \( P_{fo} \), and is denoted as \( D_{vfo} \). As a result, the total computing time is a function defined as follows:

\[
f(n^2, \sum D_{vP_v} / n_v, \sum D_{vfo} / n_{iv})
\]

where \( n^2 \) represents the number of input cells, \( \sum D_{vP_v} / n_v \) denotes the average distance from \( v \) to all of the visible cells, and \( \sum D_{vfo} / n_{iv} \) denotes the average distance from \( v \) to all of the first obstruction cells. Furthermore, the total computing time is positively correlated with each of these three factors. In Section 5, a set of experiments shows variations of computing time with different viewpoints specified.

5 Computational experiments
A suite of computational experiments was designed to evaluate computational performance of our parallel computing approach to viewshed analysis of large grid terrain datasets.

We compared among our two GPU-based parallel algorithms and the R3 sequential algorithm. The two GPU-based algorithms include GPU-fine that is described in Section 4, and GPU-coarse that is developed as a baseline to provide effective evaluation of performance gains by the fine-scale spatial domain decomposition strategy. Different from GPU-fine, the GPU-coarse algorithm does not exploit on-chip shared memory or fine-scale spatial domain decomposition strategy. To facilitate fair comparisons, all three algorithms are based on the same traversal algorithm described in Section 2.1.2. Single floating-point precision is used for representing elevation features, and computing intersections.

**Programming language and computing environment:** R3, GPU-fine and GPU-coarse were written using the standard ANSI C programming language and CUDA SDK version 4.0, and they were compiled with optimization level –O2. All were tested on the Accelerator Cluster named AC cluster (Kindratenko et al. 2009) at the National Center for Supercomputing Applications (NCSA) where we used computing nodes consisting of a CPU (dual-core 2.4 GHz AMD Opteron, 8 GB RAM) and one NVIDIA Tesla S1070 containing 4 GT200 GPUs (each with 240 CUDA cores, 4GB global memory, 16KB shared memory and 16KB 32-bit register files per SM and CUDA compute capability 1.3). We ran experiments on a single-core of CPU and a single GPU card. To ensure fair comparisons among the three algorithms, we assured that both CPU main memory and GPU global memory have 4GB capacity.
Datasets: Seven terrain datasets (Figure 11) were used. These datasets cover the state of Utah in the USA, and are 1-meter raster sampled from the 30-meter National Elevation Dataset (NED) provided by the US Geologic Survey (http://ned.usgs.gov/), using ArcGIS 10 (ESRI 2011) bilinear sampling tool. These datasets represent various sizes of terrains ranging from 400 million to over 5 billion cells (Table I), and all cells in each dataset have valid elevations. These datasets were chosen because they include mountainous areas that provide a variety of topographic features (e.g., pits, peaks, ridge, and ravine (Lee 1991)) for different viewpoint selections, making them ideal for comparing how these spatial features affect computational performance.

[Figure 11 near here][Table I near here]

5.1 Performance tuning of GPU-fine and GPU-coarse

Increasing multiprocessor occupancy (see Section 2.2) can help reduce the influence of the latency of global memory access and, thus, improve computational performance, especially for memory bandwidth-bound applications such as viewshed analysis. Multiprocessor occupancy can be optimized by tuning the usage of registers and shared memory, and thread block size. Our experiments focused on tuning thread block size to maximize the multiprocessor occupancy for both GPU-coarse and GPU-fine algorithms using CUDA GPU Occupancy Calculator (NVIDIA Corporation, 2010). The tuning tests revealed 256 as the optimal numbers of threads per block for both GPU-coarse and GPU-fine algorithms.

5.2 Performance comparison among R3, GPU-fine and GPU-coarse
To observe variation of computation time over different topography, we selected three different kinds of viewpoints for each dataset. One viewpoint was a peak on the terrain, another one was a pit and the last one was selected in areas that do not have significant topographic variation.

The resulting visibility maps of \textit{GPU-coarse} and \textit{GPU-fine} were verified by comparing to those of \textit{R3}. \textit{GPU-coarse} and \textit{R3} produced the same results. There was minor difference between \textit{GPU-fine} and \textit{GPU-coarse (or R3)}, which are caused by the fine-scale decomposition requiring the computation of the starting intersected cell every time after loading one intersected sub-tile into shared memory (see Figure 9). This starting cell is computed using different floating-point operations from those in \textit{GPU-coarse} that identify each intersected cell using the traversal algorithm only (See Section 2.1.2). It should be noted that due to difference in floating-point calculations the intersected cells identified by \textit{GPU-fine} and \textit{GPU-coarse} could be slightly different, resulting in not exactly the same visibility maps. To quantify such difference, we tested all seven datasets for \textit{GPU-fine} and \textit{GPU-coarse} based on the 3 selected viewpoints, and found the average difference is smaller than 0.00002%.

Table II summarizes computation times of the three algorithms for the seven datasets. Overall, benefiting from our parallel computing strategies, \textit{GPU-fine} greatly reduces the accumulative time span of analyses, comparing to \textit{R3} and \textit{GPU-coarse}. Both \textit{GPU-fine} and \textit{GPU-coarse} are capable of processing sizeable datasets that \textit{R3} cannot handle, which is enabled by our domain decomposition and moving window strategies. For instance, when dataset is 20.54GB, 5 times of memory capacity 4GB, our parallel algorithms can still finish computation within reasonable time.
5.2.1 Computing performance

Computing times of the three algorithms vary on three different viewpoints selected and input data size \( n \) (Table II). Given the same \( n \), the computing time for a peak viewpoint is larger than that for a pit viewpoint, because selecting a peak viewpoint usually implies larger \( n_v \) or possibly larger \( D_{ov} \) than selecting a pit viewpoint based on equation (5).

Overall, our experiments are consistent with equation (5).

The computing time speedup of \textit{GPU-fine} compared against \textit{R3} for datasets that are smaller than 4 GB is calculated to highlight the computing performance improvement contributed by massively parallel processing on GPUs. The results showed significant performance improvement, larger than 50 times speedups, were achieved (Figure 12).

5.2.2 Memory performance

When datasets are large (>4GB) and cannot be fit into global memory all at once (Figure 13(B)), up to 3 times speedups were achieved by comparing the computing time of \textit{GPU-fine} against \textit{GPU-coarse} (Figure 14), although there is no obvious speedup observed when datasets are small (<4GB) and can be processed in global memory (Table 2).

The speedups demonstrate memory performance gains of \textit{GPU-fine} from the fine-scale spatial domain decomposition strategy is more significant when datasets are larger, primarily due to the fact that reads from global memory likely perform worse when large datasets are treated. The reason for the worse performance is when as shown in Figure 13
(B), datasets are large, the coarse-scale decomposition is applied, and tiles are stored in different chunks of global memory. Consequently, intersected cells to which consecutive threads access are less likely to be contiguously stored in global memory than when processing smaller datasets (Figure 13 (A)). Therefore, when input grid is larger, the worse global memory access leads to more performance loss, and necessitates the application of the fine-scale spatial domain decomposition strategy (i.e. implemented in the GPU-fine algorithm) to improve memory access performance. Overall, the results of memory performance evaluation demonstrated the effectiveness of our fine-scale spatial domain decomposition strategy, and desirable performance of using on-chip shared memory.

[Figure 13 near here][Figure 14 near here]

6 Concluding discussion and future work

The primary contribution of this research includes a cohesive set of spatially explicit strategies for using GPUs to address data-intensive viewshed analysis. In particular, a parallel computing approach is developed to compute the viewshed of large grid terrain datasets. Specifically, our GPU-based parallel approach encompasses four interrelated major strategies: coarse-scale spatial domain decomposition, tile-based moving window, fine-scale spatial domain decomposition and massively parallel processing. These strategies have greatly enhanced memory access performance and empowered the geospatial analysis that would otherwise be considered infeasible. In doing so, the spatial characteristics of the underlying R3 algorithm and GPU parallel computing architecture features are synergistically taken into account by our approach.
Computational experiments were conducted on a supercomputer to evaluate the strategies. More than 50 times of speedups over the R3 sequential algorithm were gained for datasets of modest sizes. Furthermore, the coarse-scale spatial domain decomposition and moving window strategies make our approach scalable to large datasets that the sequential algorithm cannot handle. The experiments also demonstrated that while global memory can achieve good performance in some scenarios where memory access can make full use of its high-bandwidth, in most cases, the utilization of fast on-chip shared memory reaps superior performance.

GPUs are particularly useful for spatial data analyses that often have significant data parallelism to potentially take advantage of many-core resources and dedicated high-bandwidth memory. While this study focuses on GPU-based parallel viewshed analysis, the strategies (for regularizing memory access, reusing transferred data, and exploiting massive spatial parallelism) are designed to address generic cases of data-intensive spatial analysis. These strategies provide guiding principles to effectively extract spatial patterns of data intensity and efficiently map such patterns to underlying modern parallel computer architecture for high quality spatial analysis results and desirable computational performance. Furthermore, our approach is portable to any computer system (e.g., personal or high performance computer) equipped with CUDA-enabled GPUs.

Our ongoing work investigates the integration of our GPU-based algorithms into a cyberinfrastructure-based geospatial problem-solving environment (Wang, S. 2010). This integration will allow data-intensive viewshed analysis to be used as a set of application services and accessed simultaneously by a large number of users through reaping the benefits of dynamically harnessing on-demand cyberinfrastructure resources. In our
future work, we will investigate how to further optimize our strategies, for example, for reducing the cost for synchronization of threads; and study how to adapt our strategies to other sequential algorithms for viewshed analysis such as the radial sweep algorithm (Van Kreveld 1996; Haverkort et al. 2009; Fishman et al. 2009).

References


NVIDIA Corporation, 2010. NVIDIA CUDA™ computing guide.


Table and Figure List

Table I. Size of terrain data sets

Table II. Computation time of R3, GPU-coarse and GPU-fine for 7 datasets. Note: VP stands for viewpoint, Comp denotes computing time, and All is total running time. All the numbers in the table are rounded to nearest integers, and “h” indicates time is in hours. Throughout the paper, “pit” represents a point in a valley; “peak” is a point at the top of a mountain and “flat” denotes a point selected from an area that does not have significant topographic variation.

Figure 1. (A) Grid terrain T; (B) Domain D, shaded cells are intersected cells

Figure 2. Amanatides and Woo’s traversal algorithm: (A) initialization; (B) incremental traversal

Figure 3. An architectural illustration of NVIDIA CUDA-enabled GPUs (NVIDIA Corporation, 2009)

Figure 4. Major steps of R3 sequential algorithm

Figure 5. Coarse-scale spatial domain decomposition. Note: Dark shaded square is a target tile, and light shaded squares are intersected tiles of the target tile.

Figure 6. An example of moving tiles to and from GPU global memory. Note: For a new target tile, light shaded color represents its required tiles that are already in global memory, white color denotes a new required tile transferred to memory, and dark shaded color shows an unrequired tile removed from global memory.
Figure 7. Fine-scale spatial domain decomposition. **Note:** Squares with bold boundary are tiles; a dark shaded square represents a target sub-tile, and light shaded squares are intersected sub-tiles of the target sub-tile.

Figure 8. An example of mapping data block and thread block: one thread processes one target cell. **Note:** Light shaded color represents a thread, and dark shaded color denotes target cells that the thread processes. \( BlockID.y \) and \( BlockID.x \) are row and column indices of thread blocks in a grid respectively. \( threadID.y \) and \( threadID.x \) are row and column indices of threads in a thread block. \( BlockDim.y \) and \( BlockDim.x \) are the number of rows and columns in a thread block. \((BlockID.y, BlockID.x, threadID.y, threadID.x)\) uniquely identifies a thread. \((row, col)\) are row and column indices of target cells in a data block.

Figure 9. Finding the starting intersected cell from the intersected sub-tile in shared memory for a target cell. **Note:** Assuming that the origin \((0, 0)\) of an Euclidian coordinate system is at the top left of an input grid, \((x, y)\) is coordinates of the starting intersection of LOS ray with the intersected sub-tile, \((X_{\text{start}}, Y_{\text{start}})\) is the column and row indices of the starting intersected cell in the grid.

Figure 10. Pseudo code for the working process of a thread block

Figure 11. Test datasets used for the experiments: the bounding boxes of 7 test datasets are shown in green

Figure 12. Speedup = \( \text{GPU-fine} \) computing time / \( R3 \) computing time, varying two datasets and three selected viewpoints

Figure 13. \( \text{GPU-coarse} \) global memory access patterns: (A) when input data can fit into global memory; (B) when input data cannot fit into global memory. **Note:** Squares with
bold boundary are tiles, dark shaded color represents target cells processed by consecutive threads, and light shaded color shows intersected cells required by the consecutive threads at any point in time.

Figure 14. Speedup = \textit{GPU-coarse} computing time / \textit{GPU-fine} computing time, varying five datasets and three selected viewpoints
Table I. Size of terrain data sets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Cells</th>
<th>Size (bytes)</th>
<th>Maximum elevation (meter)</th>
<th>Minimum elevation (meter)</th>
<th>Average elevation (meter)</th>
<th>Standard deviation (meter)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20000×20000</td>
<td>1.49G</td>
<td>1900.29</td>
<td>1419.99</td>
<td>1467.29</td>
<td>32.69</td>
</tr>
<tr>
<td>2</td>
<td>28000×28000</td>
<td>2.92G</td>
<td>2803.43</td>
<td>1410.19</td>
<td>1559.78</td>
<td>195.52</td>
</tr>
<tr>
<td>3</td>
<td>33280×33280</td>
<td>4.13G</td>
<td>2951.91</td>
<td>1407.10</td>
<td>1617.78</td>
<td>247.68</td>
</tr>
<tr>
<td>4</td>
<td>46080×46080</td>
<td>7.91G</td>
<td>2951.91</td>
<td>1387.14</td>
<td>1659.92</td>
<td>261.92</td>
</tr>
<tr>
<td>5</td>
<td>57024×57024</td>
<td>12.11G</td>
<td>2951.91</td>
<td>1387.14</td>
<td>1668.93</td>
<td>244.98</td>
</tr>
<tr>
<td>6</td>
<td>65280×65280</td>
<td>15.88G</td>
<td>2951.91</td>
<td>1387.14</td>
<td>1712.70</td>
<td>277.58</td>
</tr>
<tr>
<td>7</td>
<td>74256×74256</td>
<td>20.54G</td>
<td>3082.65</td>
<td>1387.14</td>
<td>1764.89</td>
<td>316.46</td>
</tr>
</tbody>
</table>
Table II. Computation time of $R3$, $GPU$-coarse and $GPU$-fine for 7 datasets. Note: $VP$ stands for viewpoint, $Comp$ denotes computing time, and $All$ is total running time. All the numbers in the table are rounded to nearest integers, and “h” indicates time is in hours.

Throughout the paper, “pit” represents a point in a valley; “peak” is a point at the top of a mountain and “flat” denotes a point selected from an area that does not have significant topographic variation.

<table>
<thead>
<tr>
<th>Dataset (bytes)</th>
<th>$VP$</th>
<th>$R3$ (second)</th>
<th>$GPU$-coarse (second)</th>
<th>$GPU$-fine (second)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>$Comp$</td>
<td>$All$</td>
</tr>
<tr>
<td>1.49G</td>
<td>Pit</td>
<td>16917</td>
<td>203</td>
<td>229</td>
</tr>
<tr>
<td></td>
<td>Flat</td>
<td>16642</td>
<td>293</td>
<td>319</td>
</tr>
<tr>
<td></td>
<td>Peak</td>
<td>24971</td>
<td>354</td>
<td>380</td>
</tr>
<tr>
<td>2.92G</td>
<td>Pit</td>
<td>35h</td>
<td>903</td>
<td>997</td>
</tr>
<tr>
<td></td>
<td>Flat</td>
<td>60h</td>
<td>920</td>
<td>1014</td>
</tr>
<tr>
<td></td>
<td>Peak</td>
<td>85h</td>
<td>2118</td>
<td>2212</td>
</tr>
<tr>
<td>4.13G</td>
<td>Pit</td>
<td>2526</td>
<td>7009</td>
<td>7079</td>
</tr>
<tr>
<td></td>
<td>Flat</td>
<td>7224</td>
<td>5028</td>
<td>5186</td>
</tr>
<tr>
<td></td>
<td>Peak</td>
<td>12616</td>
<td>12769</td>
<td>5248</td>
</tr>
<tr>
<td>7.91G</td>
<td>Pit</td>
<td>1124</td>
<td>1179</td>
<td>1433</td>
</tr>
<tr>
<td></td>
<td>Flat</td>
<td>5028</td>
<td>14462</td>
<td>14759</td>
</tr>
<tr>
<td></td>
<td>Peak</td>
<td>12616</td>
<td>22776</td>
<td>23003</td>
</tr>
<tr>
<td>12.11G</td>
<td>Pit</td>
<td>799</td>
<td>6159</td>
<td>6921</td>
</tr>
<tr>
<td></td>
<td>Flat</td>
<td>14462</td>
<td>12707</td>
<td>13627</td>
</tr>
<tr>
<td></td>
<td>Peak</td>
<td>16586</td>
<td>16586</td>
<td>17660</td>
</tr>
<tr>
<td>15.88G</td>
<td>Pit</td>
<td>353</td>
<td>6159</td>
<td>6921</td>
</tr>
<tr>
<td></td>
<td>Flat</td>
<td>11171</td>
<td>12707</td>
<td>13627</td>
</tr>
<tr>
<td></td>
<td>Peak</td>
<td>16586</td>
<td>16586</td>
<td>17660</td>
</tr>
<tr>
<td>20.54G</td>
<td>Pit</td>
<td>26030</td>
<td>26030</td>
<td>28044</td>
</tr>
</tbody>
</table>
Figure 1. (A) Grid terrain $T$; (B) Domain $D$, shaded cells are intersected cells
Figure 2. Amanatides and Woo’s traversal algorithm: (A) initialization; (B) incremental traversal

```
Loop (before the end of line or until the target cell may be determined as visible) {
    if(tVertical < tHorizontal) {
        tVertical += tDeltaV;
        X = X + stepX;
    } else {
        tHorizontal += tDeltaH;
        Y = Y + stepY;
    }
    ProcessCell(X,Y);
}
```
Figure 3. An architectural illustration of NVIDIA CUDA-enabled GPUs (NVIDIA Corporation, 2009)
Figure 4. Major steps of R3 sequential algorithm

When grid terrain data are large, it cannot be held in memory at once.

- Step 1: Read grid terrain data and the location of v into memory
- Step 2: Find intersected cells along LOS for each target cell
- Step 3: Compute visibility for each target cell
- Step 4: Output viewshed results

Computationally intensive
Figure 5. Coarse-scale spatial domain decomposition. **Note:** Dark shaded square is a target tile, and light shaded squares are intersected tiles of the target tile.

**Proof** If there is at least one tile completely included between two lines. Then, $b_1/b_2 = l_1/l_2$ where $b_1 = b_2$, so $l_1 = l_2$. However, $l_1 < l_2$, it is contradicted. Therefore, there is no tile completely occluded by two lines from v to the start point and the end point of a target tile.
Figure 6. An example of moving tiles to and from GPU global memory. **Note:** For a new target tile, light shaded color represents its required tiles that are already in global memory, white color denotes a new required tile transferred to memory, and dark shaded color shows an unrequired tile removed from global memory.
Figure 7. Fine-scale spatial domain decomposition. Note: Squares with bold boundary are tiles; a dark shaded square represents a target sub-tile, and light shaded squares are intersected sub-tiles of the target sub-tile.
Figure 8. An example of mapping data block and thread block: one thread processes one target cell. **Note:** Light shaded color represents a thread, and dark shaded color denotes target cells that the thread processes. $BlockID.y$ and $BlockID.x$ are row and column indices of thread blocks in a grid respectively. $threadID.y$ and $threadID.x$ are row and column indices of threads in a thread block. $BlockDim.y$ and $BlockDim.x$ are the number of rows and columns in a thread block. $(BlockID.y, BlockID.x, threadID.y, threadID.x)$ uniquely identifies a thread. $(row, col)$ are row and column indices of target cells in a data block.
Figure 9. Finding the starting intersected cell from the intersected sub-tile in shared memory for a target cell. **Note:** Assuming that the origin \((0, 0)\) of an Euclidian coordinate system is at the top left of an input grid, \((x, y)\) is coordinates of the starting intersection of LOS ray with the intersected sub-tile, \((X_{start}, Y_{start})\) is the column and row indices of the starting intersected cell in the grid.
Kernel program for each thread

__shared__ volatile Boolean flag

..........

while the intersected cells in the grid are not traversed to the end
    __syncthreads()    // all the threads in the block are synchronized
    flag = 0
    __syncthreads()

    if the target cell may be visible
    then flag = 1

    ..........

    __syncthreads()
    if flag == 1
        then load the next intersected sub-tile into shared memory
            while the intersected cells in the intersected sub-tile is not traversed to the end
                if the target cell is not invisible
                    then access its intersected cells in the shared memory and compute its visibility
                    otherwise break
            otherwise break

Figure 10. Pseudo code for the working process of a thread block
Figure 11. Test datasets used for the experiments: the bounding boxes of 7 test datasets are shown in green
Figure 12. Speedup = $GPU$-fine computing time / $R3$ computing time, varying two datasets and three selected viewpoints
Figure 13. *GPU-coarse* global memory access patterns: (A) when input data can fit into global memory; (B) when input data cannot fit into global memory. **Note:** Squares with bold boundary are tiles, dark shaded color represents target cells processed by consecutive threads, and light shaded color shows intersected cells required by the consecutive threads at any point in time.
Figure 14. Speedup = $GPU$-coarse computing time / $GPU$-fine computing time, varying five datasets and three selected viewpoints