APPLICATIONS PAPER

GPU-Accelerated Multiple Observer Siting

Wenli Li, W. Randolph Franklin, Salles Viana Gomes de Magalhães, and Marcus V. A. Andrade

Abstract

We present two fast parallel implementations of the Franklin-Vogt multiple observer siting algorithm, using either OpenMP or CUDA. In this problem, the objective is to site observers on a raster terrain such that the joint area visible by them is maximized. On a portion of terrain with 16,385×16,385 cells, assuming that observers can see out to a radius-of-interest of 100 cells, finding the approximate 15,000 observers that have the greatest coverage takes only 17s in CUDA. That is a factor of 70 speedup from the sequential version. The OpenMP version exhibits a factor of 17 speedup on a 16 core system. Applications for the multiple observer siting problem include radio transmission towers, environmental monitoring sites, and path planning for surveillance drones. The algorithm has four steps: finding the visibility indices of all points, selecting a candidate subset of potential top observers, finding each one's viewshed, and greedily constructing the final solution.

Introduction

The purpose of multiple observer siting (Franklin, *et al.* 2002) is to place observers to cover the surface of a terrain or of targets above the terrain. It is useful in the placement of radio transmission towers, mobile ad hoc networks, and environmental monitoring sites.

As described in (Magalhães, *et al.*, 2010), an *observer* is a point in space from which we wish to see or communicate with other points, called *targets*. The usual notation for observer and target is *O* and *T*. Both the observer and the targets are considered to be at some given height above the ground (useful for modeling, e.g., towers). The base points of *O* and *T* are the points on the terrain directly below *O* and *T*, respectively.

An observer *covers* (or views) a target that is within a given distance (called *radius of interest*), and that have a direct line of sight (LOS) from the observer. Figure 1 illustrates these concepts using a section of a terrain model.

The *viewshed V* of an observer *O* is the set of all terrain points (base points) where targets positioned on these points are visible from *O*. The *visible area* of an observer is the size of its viewshed.

The *joint viewshed* of a set of terrain points is the union of the individual viewsheds of observers sited above these points. Similarly, the *joint visible area* (or simply *visible area*) of a set of terrain points is the size of its joint viewshed.

Given a terrain represented as a digital elevation matrix (DEM), there are many choices of what to optimize with the observer siting. For example, the objective may be to select a fixed number of observers whose visible area is maximized. Alternatively, the problem may be to select as few points as possible such that their visible area is at least a given threshold value.

Wenli Li and W. Randolph Franklin are with Rensselaer Polytechnic Institute, Troy, New York (mail@wrfranklin.org).

Salles Viana Gomes de Magalhães is with Rensselaer Polytechnic Institute, Troy, New York, and the University Federal de Viçosa, Viçosa, MG, Brazil.

Marcus V. A. Andrade is with Federal de Viçosa, Viçosa, MG, Brazil.

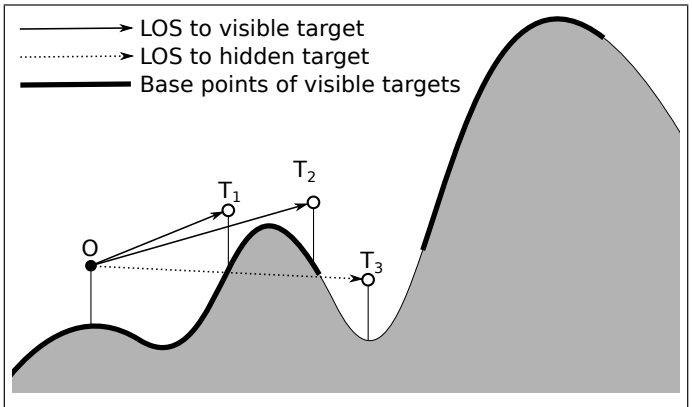

Figure 1. Visibility of targets from observers (adapted from (Magalhães, S.V., *et al.*, 2010)).

Other variations of the problem include: using observers that can cover targets with a quality or probability (Akbarzadeh, *et al.*, 2013), adding constraints such as intervisibility, where observers are required to be visible from other observers, using mobile observers and targets (Efrat, *et al.*, 2012), and having different costs for placing observers at different positions.

The terrain may be either the traditional surface of the earth, or the tops of buildings in an urban terrain. The observers may be in remotely operated or autonomous airborne vehicles, whose operators desire to optimize the flight path. This may require repeatedly re-computing the siting problem with slightly varying parameters.

A recent application requiring optimizing observer siting is *Li-Fi*, or *light fidelity*, which switches LEDs on and off at a high speed to effect high speed communication. Its main advantage is its immunity to electronic interference. Its visibility computation is complicated by its light's ability to reflect from shiny surfaces.

"Visibility" is also applicable to GHz radio signals that can be blocked by heavy objects such as reinforced concrete pillars. That is relevant to siting the thousands of beacons that may be required for indoor way-finding in large buildings such as airports. For example, Schipol airport's way-finding system has 2000 beacons.

Other notable related research includes the many line-of-sight issues in the Modeling and Simulation community discussed (with comparisons of various LOS algorithms) (Line-Of-Sight Technical Group, 2004), the relation of visibility to topographic features (Lee, 1992), and the pioneering work of (Nagy, 1994). (Champion, *et al.*, 2002) studied line-of-sight on natural terrain defined by an L_1-spline.

Multiple observer siting is a compute-intensive problem with considerable inherent parallelism. That, plus the recent

Photogrammetric Engineering & Remote Sensing
Vol. 83, No. 6, June 2017, pp. 439–446.
0099-1112/17/439–446

doi: 10.14358/PERS.83.6.439

availability of higher resolution geographical databases, makes it a good candidate for parallelization. In particular, the multiple observer siting algorithm of (Franklin and Vogt, 2004) and (Franklin and Vogt, 2006) can be optimized to reduce its time and space complexity. Parallelizing the algorithm would greatly increase its speed, so that the program would be very fast for small terrains and useful for large ones.

Recently, the parallel computing capabilities of GPUs have attracted increasing interest of researchers. These devices contain thousands of processing units and, even though they were designed for computer graphics applications, they are fully programmable and optimized for SIMD vector operations. Using GPUs for scientific computing is called general-purpose computing on graphics processing units (GPGPU) (Luebke, *et al.*, 2004). Although a CPU core is over ten times more powerful than a GPU core, a GPU may have over 2,000 cores, while a high-end CPU might have only 28 cores. Recent GPUs have up to a few hundred GB/s of memory bandwidth and a few TFLOPS of single-precision processing power, although achieving theoretical peak performance is nontrivial.

The parallelization of line-of-sight and viewshed algorithms on terrains using GPGPU or multi-core CPUs is an active topic. (Strnad, 2011) parallelized the line-of-sight calculations between two sets of points (a source set and a destination set) on a GPU, and implemented it on a multi-core CPU for comparison. (Zhao, *et al.*, 2013) parallelized the R3 algorithm (Franklin and Clark, 1994) to compute viewsheds on a GPU. The parallel algorithm combines coarse-scale and fine-scale domain decompositions to deal with memory limit and enhance memory access performance. (Osterman, 2012) parallelized the *r.los* module (R3 algorithm) of the open-source GRASS GIS on a GPU. (Osterman, *et al.*, 2014) also parallelized the R2 algorithm (Franklin, and Clark, 1994). (Axell, and Mattias, 2015) parallelized and compared the R2 algorithm on a GPU and on a multi-core CPU. (Bravo, *et al.*, 2015) parallelized the XDRAW algorithm (Franklin and Clark, 1994) to compute viewsheds on a multi-core CPU, after improving its IO efficiency and compatibility with SIMD instructions. (Ferreira, *et al.*, 2014) and (Ferreira, *et al.*, 2016) parallelized the sweep-line algorithm of (Kreveld, 1996) to compute viewsheds on multi-core CPUs. In multiple observer siting, (Magalhães, *et al.*, 2010) proposed a local search heuristic to increase the percentage of a terrain covered by a set of k observers. Given a set of candidate observers, each subset of k observers is a solution S, and each solution with one observer different from S is a neighbor of S. Starting from an initial solution, the heuristic repeatedly replaces the current solution with a better neighbor, until a local optimum is found. (Pena, *et al.*, 2014) improved the performance of the heuristic by accelerating the overlay of viewsheds on a GPU using dynamic programming and a sparse-dense matrix multiplication algorithm.

In this paper, we optimize and parallelize the multiple observer siting algoirthm of Franklin and Vogt for GPUs. We also implement it on multi-core CPUs for comparison. In the next section, the sequential multiple observer siting algorithm will be reviewed.

Multiple Observer Siting

(Franklin and Vogt, 2004) and (Franklin and Vogt, 2006) proposed an algorithm to select a set of observers to cover the surface of a terrain. Let the *visibility index* of a terrain point be the visible area of an observer sited on the point divided by the total number of terrain points within an observer radius of interest. The algorithm first computes an approximate visibility index for each terrain point, and selects a set of terrain points with high visibility indexes as candidate observer positions. The set of observers at the candidate positions are called *tentative observers*. In the next step, the viewshed of each tentative observer is computed and an iterative process is employed to greedily select observers from the set of tentative observers to cover the terrain surface. As an option, the algorithm can be configured to only select observers that are visible from one of the other selected observers (intervisibility). At the top level, this algorithm has four sequential steps: VIX, FINDMAX, VIEWSHED, and SITE.

VIX, the first step, computes an approximate visibility index for each terrain point, which is normalized as an integer from 0 to 255 and stored in a byte. The parameters of VIX include the number of rows or columns of the terrain (*nrows*), the radius of interest of observers (*roi*), the height of observers or targets above the terrain (*height*), and the number of random targets for each observer. For simplicity, the algorithm assumes that the terrain is a square and the observer height and the target height are equal, but these restrictions are easy to remove. Placing an observer at each terrain point, VIX picks a number of random terrain points within the radius of interest as targets and calculates their line-of-sight visibility. Then it calculates the ratio of visible targets, normalized in 0-255, as the approximate visibility index of the terrain point.

FINDMAX, the second step, selects a small subset of terrain points with high visibility indexes (the *tentative observers*). Since highly visible points are often close to each other, for example, along ridges and on water surfaces, FINDMAX divides the terrain into square blocks and selects a number of highly visible points in each block as tentative observers. The parameters of FINDMAX include the number of observers to select and the block width.

VIEWSHED, the third step, computes the viewshed of each tentative observer using the R2 algorithm. With a square of width (*2roi+1*) centered at an observer, the algorithm shoots a ray from the observer to each point on the boundary of the square and calculates the visibility of the points along the rasterized ray. Terrain points within *roi* along a ray are visited in order and their visibility is determined by the local horizon. The observer and the points as targets are *height* above the terrain, while the points as obstacles are on the terrain. Figure 2 shows a schematic illustration of the algorithm with *roi* = 3. On the top of the figure, an observer O is at the center of a 7 × 7 cells square, wherein the shaded point cells are not within *roi* and the dashed lines are the rays from O to the boundary points. On the bottom of the figure, the points along rays OA and OB are on a 1D terrain and their visibility is determined by the local horizon (dotted lines). Because of the discrete nature of the algorithm, a point is often on multiple rays and is considered visible if it is visible along any ray.

SITE, the last step, selects a set of observers from the tentative observers to cover the terrain. It uses a greedy algorithm that selects one tentative observer at a time. SITE keeps two main data structures: S, the resulting set of the observers selected during the siting process and C, the union of the viewsheds (the *joint viewshed*) of the observers in S. Initially both S and C are empty. In each iteration SITE selects the unused tentative observer whose viewshed would maximize the size of C. I.e., for each unused tentative observer O, the number of cells in the union of C with the viewshed of O is computed and the observer that maximizes this number is selected. Once an observer is selected it is inserted into S and C is updated. This process is accelerated by representing the viewsheds (and the joint viewshed) with bit matrices (where 1 and 0 represent, respectively, visible and non-visible cells), and using bitwise OR and population count operations to compute, respectively, the union and the visible area of viewsheds. The siting process may stop when a maximum number of sited observers is achieved or when the size of C reaches a minimum target value.

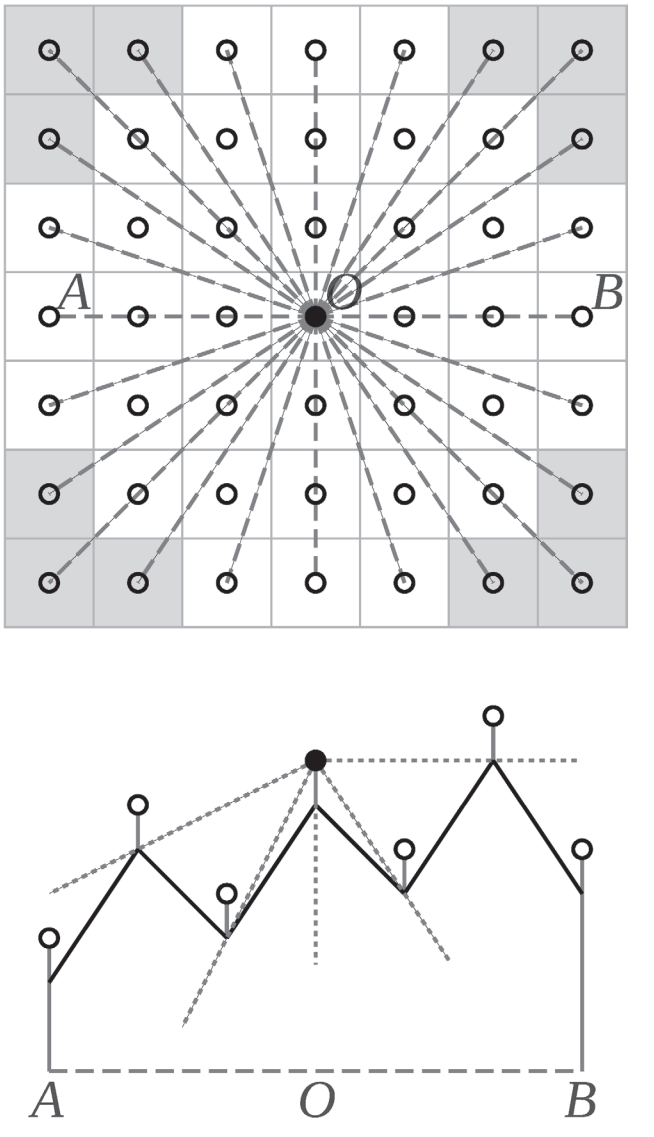

Figure 2. Computing a viewshed using the R2 algorithm.

Performance Optimization

Before parallelizing the observer siting algorithm, it was first optimized to reduce its time and space cost. Minimizing space is important for GPU parallel programs because the GPU memory is usually much smaller than the CPU memory, and memory copies between them are expensive. In this section we will describe how each step of the observer siting algorithm was optimized.

VIX: this is often the most time-consuming step. As it computes approximate visibility indexes by target sampling, it can compute approximate line-of-sight visibilities by processing only some sample points along the line to further increase speed. It still uses random points within *roi* of an observer as targets. (Rana, 2003) proposed using topographic feature points as targets, but it is not definitely better than using random points. For each target, instead of computing a line-of-sight visibility by evaluating all points along the line between the observer and the target, VIX computes an approximate line-of-sight visibility by evaluating a subset of the points with an *interval* between successive points of evaluation along the line of sight. The idea is like volume ray casting in Computer Graphics, which uses equidistant sampling points along a viewing ray for shading and compositing. *interval* = 1 is equivalent to evaluating all the points. The choice of *interval* and its effects on the outcome will be discussed in the Results Section.

FINDMAX: instead of selecting multiple tentative observers per block, we modified FINDMAX to select one tentative observer per block. This strategy has two benefits. First, selecting one tentative observer is accomplished by scanning the block points for the highest visibility index, while selecting multiple tentative observers requires sorting the points by visibility index. Second, preliminary experiments have shown that points with high visibility indices within a block tend to be close to each other, which may reduce the quality of the results since nearby observers usually have viewsheds covering redundant areas. By selecting only one tentative observer per block and reducing the size of the blocks (in order to not change the total number of tentative observers), we make the tentative observers more evenly distributed throughout the terrain, and reduce the time complexity of FINDMAX to be linear in the terrain size, i.e., $\Theta(nrows^2)$.

VIEWSHED: since R2 was the viewshed algorithm initially employed in the observer siting algorithm, and it presents a good balance of speed and accuracy, we decided to continue using R2 in the optimized version of our siting algorithm. However, performing experiments with other viewshed algorithms is an interesting direction for future work. For example, the XDRAW algorithm is faster but less accurate than R2. (Wang, 2000) proposed a viewshed algorithm that uses a plane instead of lines of sight in each of eight standard sectors around the observer to approximate the local horizon. The algorithm is faster but less accurate than XDRAW. (Israelevitz, 2003) extended XDRAW to increase accuracy by sacrificing speed.

Since only pixels within the radius of interest (*roi*) of the observers can be visible in a viewshed, we represented the viewsheds of observers using a matrix with $(2roi+1)\times(2roi+1)$ pixels. The joint viewshed is represented using *nrows* × *nrows* pixels. Each pixel uses only one bit and each row of a viewshed is padded to the word size. The bit representation is compact and fast to process using bitwise operators. If the rows were not padded, accessing a given cell of a viewshed would be easier but overlaying a viewshed with a joint viewshed would be harder because of misalignments between words of the two matrices. The number of tentative observers is $\frac{nrows^2}{bw^2}$ (*bw* is the block width) and the time complexity of R2 is linear in the maximum number of visible points in a viewshed ($O(roi^2)$), so that the time of VIEWSHED is $O(\frac{nrows^2}{bw^2}roi^2)$.

The time cost of SITE is highly dependent on the number of tentative observers the algorithm is configured to use. The original version of SITE (Franklin, 2002) computed in each iteration the areas of the union of each viewshed V with the joint viewshed C. We performed two modifications to accelerate the algorithm.

First, since the size of a viewshed matrix is $(2roi+1)^2$ pixels and the size of C is $nrows^2$, the time to compute the area of the union between a viewshed and C is $O(nrows^2)$. Given an observer O and its viewshed V, let $|V+C|-|C|$ be the contribution area of O (i.e., the contribution area represents how much the area of C would increase if V was united with C). Selecting the observer with a viewshed V that maximizes $|V+C|$ (area of the union) is equivalent to select the observer with the largest contribution area. To compute the contribution area of a viewshed V with respect to a joint viewshed C, the number of visible pixels in V that are labeled as non-visible in C is counted (this is performed using bitwise operations). Since the number of pixels that needs to be evaluated is

proportional to the size of the viewsheds, the time complexity of this operation is O(roi^2) instead of the O($nrows^2$) necessary to compute the area of the union.

Second, it is unnecessary to recompute the area of contribution of all unused tentative observers in each iteration of the siting algorithm. Once a joint viewshed is united with a viewshed of an observer *O*, the observers whose contribution areas will be modified are only the ones within *2roi* pixels of *O*. The number of tentative observers within *2roi* is approximately $\frac{4\pi roi^2}{bw^2}$.

In each iteration of site, the time to find the unused tentative observers whose contribution areas needs computing is O($\frac{nrows^2}{bw^2}$). The time to compute the contribution area of them is O($\frac{roi^4}{bw^2}$). The time to find the tentative observer to add (with the largest contribution area) is O($\frac{nrows^2}{bw^2}$). The time to update C is O(roi^2). Summing all up, the time complexity of an iteration of site is O($\frac{nrows^2}{bw^2}+\frac{roi^4}{bw^2}+roi^2$). If O($\frac{roi^4}{bw^2}$)=O(1), which is reasonable, the time complexity is O($\frac{nrows^2}{roi^2}+roi^2$). Algorithm 1 shows the observer siting algorithm. It was designed to stop once a target coverage area is obtained (or when the contribution area of all the unused tentative observers is 0).

Algorithm 1: The SITE algorithm

Input: A DEM, a set of tentative observers, and their viewsheds
Output: A set of observers
the set of observers and the joint viewshed are empty;
while coverage is lower than a threshold and can be increased **do**
foreach unused tentative observer O **do**
if the set of observers is empty or O is within 2roi of the last added observer **then**
compute the contribution area of O;
end
end
add the unused tentative observer with the largest contribution area to the set of observers;
update the joint viewshed as its union with the viewshed of the newly added observer;
end
return the set of observers;

Parallelization

The multiple observer siting algorithm is compute-intensive, but with considerable inherent parallelism. We parallelized each of the four steps of the algorithm using the Compute Unified Device Architecture (CUDA) (NVIDIA, 2016), a parallel computing platform and programming model for NVIDIA GPUs.

An NVIDIA GPU may have tens of streaming multiprocessors, each with hundreds of CUDA cores. To perform a task using CUDA, it is necessary to define a kernel function that is applied to multiple data in parallel by a large number of threads (SIMD). Each thread does a fraction of the work and is executed on one CUDA core. The threads are grouped into thread blocks and each thread block is executed on a multiprocessor. The threads of a block are divided into warps of 32 threads and each warp is instruction locked. The threads of a block can be synchronized, while the threads of different blocks cannot be synchronized, or even communicate.

VIX, the first step, computes a visibility index for every terrain point. The number of points is usually very large, so that we define a kernel function to compute the visibility index of a point in each CUDA thread. The function picks random targets for the point and calculates their line-of-sight visibility.

FINDMAX, the second step, finds the point with the highest visibility index in every terrain block as a tentative observer. The number of terrain blocks is usually much smaller than the number of terrain points. We define a kernel function to find the most visible point in a terrain block in each thread block. The function finds the most visible point in a portion of the terrain block associated to each thread block. Since multiple threads are assigned to process a terrain block, a final parallel reduction is employed to find the most visible point among the threads.

VIEWSHED, the third step, computes the viewshed of every tentative observer. The number of tentative observers is the same as the number of terrain blocks (the target points in the viewshed are not restricted to be the ones inside each terrain block). We define a kernel function such that each viewshed will be computed collectively by the threads of a thread block (each thread computes a sector of the viewshed). For example, if the block has four threads, then each thread computes a quadrant of the viewshed. In the case of Figure 2, each thread shoots six consecutive rays from the observer to points on the boundary of the square in order to compute one fourth of the viewshed.

The major task in an iteration of SITE, the last step, is to compute the contribution area of unused tentative observers within 2*roi* of the last added observer. In an earlier attempt, we defined a kernel function to check if the contribution area of a tentative observer needs to be recomputed and, if so, recompute it. The function is slow because the contribution area of most tentative observers does not need to be recomputed, which leads to a workload unbalance as most of the threads do not need to recompute the contribution area while the GPU has a SIMD architecture. We employed the following strategy to improve the workload balance: the contribution area computation process was split into two subtasks. First, a kernel determines in parallel what tentative observers need to have their contribution area recomputed. Second, a separate kernel was defined to recompute the contribution areas of the observers selected by the first kernel. To better explore the GPU parallel capability, this second kernel was developed such that each thread in a thread block processes one or more rows of the viewshed of a given observer.

A third kernel function is defined to use a parallel reduction in order to find the unused tentative observer with the largest contribution area among the tentative observers being processed by each thread block. After this kernel is applied, a CPU code selects the tentative observer with the largest contribution area among the ones chosen by the GPU code for each thread block.

Finally, a fourth kernel function is defined to update the joint viewshed. Given the viewshed of the tentative observer selected in the previous step, each thread unites one row of the viewshed with the joint viewshed. Since viewsheds are typically too small, this kernel does not fully utilize the GPU. However, doing this processing on the CPU would be slower.

To sum up, the CUDA program has seven kernel functions, one each for VIX, FINDMAX, and VIEWSHED, and four for SITE. The diagram in Figure 3 illustrates these functions.

For comparison, we parallelized the algorithm using OpenMP for executing on multi-core CPUs. OpenMP uses compiler directives and library routines to direct the parallel execution. In the OpenMP program, we used a *#pragma omp parallel for schedule(guided)* directive for parallelizing the following loops (the step of the algorithm where the loop is located is between parenthesis):

- for each terrain point, compute its visibility index (VIX)
- for each terrain block, find the most visible point as a tentative observer (FINDMAX)
- for each tentative observer, compute its viewshed (VIEWSHED)

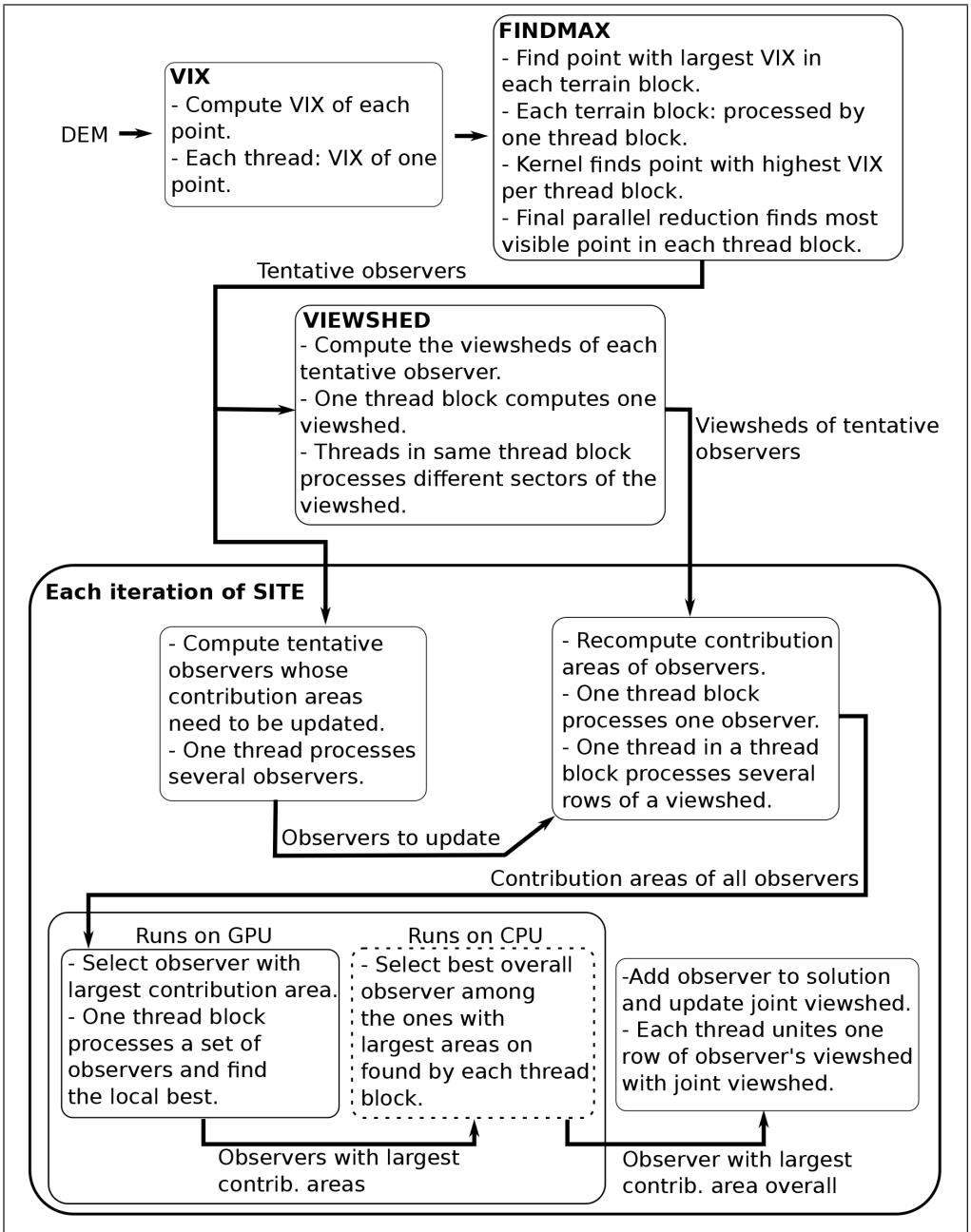

Figure 3. Diagram of the GPU observer siting algorithm.

- for each tentative observer, check if its contribution area needs to be recomputed (SITE)
- for each row of a viewshed, compute its contribution area (SITE)
- for each row of a viewshed, update the joint viewshed (SITE)

In addition, a parallel region (using a #pragma omp parallel directive) finds the unused tentative observer with the largest contribution area. Each thread finds a tentative observer in a portion of the tentative observers. At the end, each thread updates the global tentative observer with the largest contribution area using a critical region directive (#pragma omp critical).

Results and Discussion

Both the parallel and sequential implementations were tested on a machine with an NVIDIA Tesla K20Xm GPU accelerator (14 streaming multiprocessors, 192 CUDA cores per multiprocessor, and 6GB GDDR5 memory), two Intel Xeon E5-2687W CPUs at 3.1GHz (8 cores and 16 hyper-threads per CPU), and 128GB DDR3 memory, running Ubuntu 16.04 LTS. The CUDA program was compiled with nvcc 7.5.17 and gcc 4.9.3 at optimization level -O2. The OpenMP and sequential programs are compiled with gcc 4.9.3 at optimization level –O2. The terrain dataset is a 16,385 × 16,385 Puget Sound DEM from (Lindstrom, et al., 2001), which was extracted from USGS 10 meter DEMs. The unit of vertical resolution is 0.1 meter and the range of values is [0, 43930] units. Figure 4 shows a shaded relief plot of the terrain, which is about half mountains and half plains.

VIX computes an approximate line-of-sight visibility with an *interval* between successive evaluation points along each line of sight. We used the CUDA program to test the effects of *interval* because it is the fastest. To evaluate the accuracy of the approximate visibility index, the exact visibility index map of the terrain, normalized in integers 0 - 255, is computed with *roi* = 100 (1,000 meters) and observer/target *height* = 100 (10 meters). Figure 5 shows the exact visibility index map of the terrain. The figure shows that highly visible points (light colored) are on flat terrain, i.e.,plains, valleys, and water bodies. In the simplest case, *interval* can be a constant value. However, we found that points closer to the observer along the line of sight are more important for determining visibility. For example, together the eight points that are adjacent to the observer on the grid can block the visibility of all the other terrain points, while a point that is far from the observer can block the visibility of much fewer points. Therefore, evaluation points should become denser and *interval* should become smaller towards the observer. The same is true from the viewpoint of the target, so that *interval* should become smaller towards the target. We tested the CUDA program with *roi* = 100, *height* = 100, block width (*bw*) = 50 (107584 tentative observers), and target coverage = 95 percent. VIX uses 10, 50, or 250 random targets per point and different values of *interval*. Let the observer be at point 0, and the target be at point 100 along the line of sight. Starting from point 1, the evaluation points are point 1, 1+*interval*, 1+2*interval*, and so on, if *interval* is constant. The values of *interval* (and the corresponding evaluation points) in Table 1 were used in the experiments.

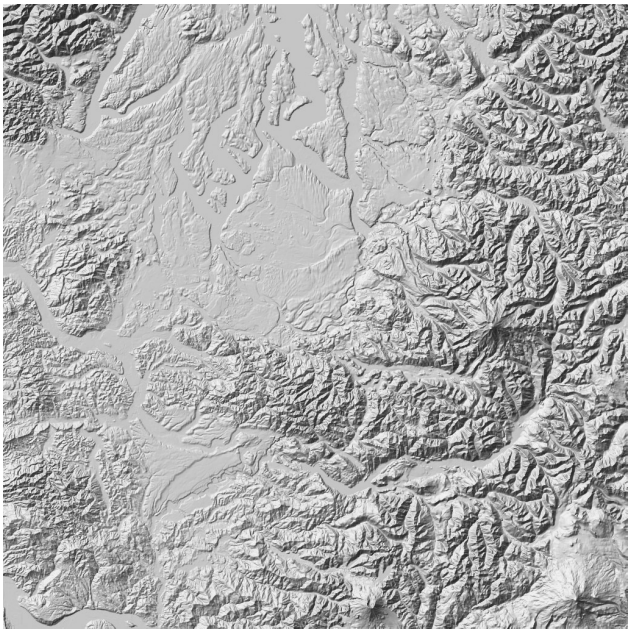

Figure 4. Shaded relief plot of the terrain dataset.

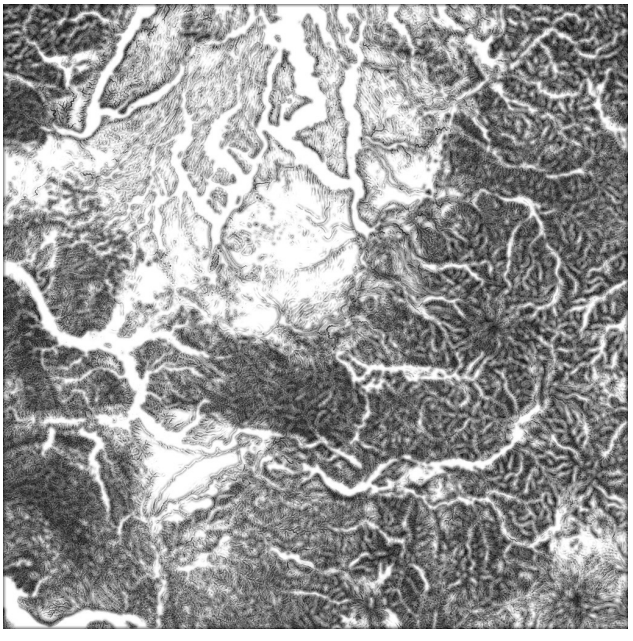

Figure 5. Exact visibility index map of the terrain, normalized in integers 0 - 255, with *roi* = 100 and *height* = 100.

Table 1. Intervals Used in the Experiments with the VIX Algorithm

Interval	# points	Points	Interval	# points	Points
1	99	1,2,3,..,99	32	4	1,33,65,97
2	50	1,3,5,..,99	Exponential	7	1,2,4,..,64
4	25	1,5,9,..,97	Fibonacci	10	1,2,3,..,89
8	13	1,9,17,..,97	Bidirect. exp.	12	1,2,4,..,96,98,99
16	7	1,17,33,..,97	Bidirect. Fib.	16	1,2,3,..,97,98,99

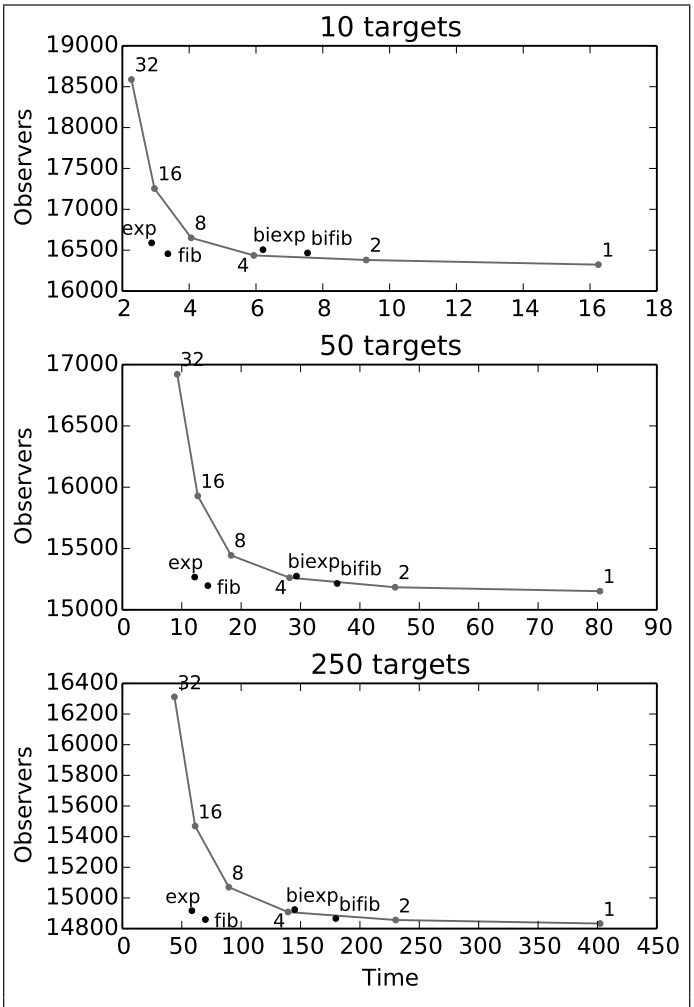

Figure 6. Running time of CUDA VIX in seconds versus the number of observers selected for 95 percent coverage. Data points are labeled with the value of *interval*.

Table 2 shows the running time of VIX in seconds, the RMSE of the approximate visibility index map (VIM), and the number of observers selected for the target coverage. The smaller the number of observers, the better the result. More random targets per point produces a longer running time for VIX, a smaller RMSE of the VIM, and a smaller number of observers. The improvement from 10 to 50 targets is larger than the improvement from 50 to 250 targets. However, a smaller RMSE does not necessarily mean fewer observers will be sited. For example, *interval* = 8 has a smaller RMSE but more observers than *interval* = 'exponential' or 'Fibonacci'. A larger *interval* produces shorter time, larger RMSE, and more observers. Figure 3 shows that (*interval* =) 'exponential' and 'Fibonacci' are below and thus better than the 1, 2, …, 32 curve, while 'bidirectional exponential' and 'bidirectional Fibonacci' are almost on the curve. We choose 50 targets and *interval* = 'exponential' for VIX, so that its time complexity is O()).

Table 3 shows the results of the parallel and sequential programs with different combinations of *roi* and *bw*: *roi* = 50 and *bw* = 25, *roi* = 100 and *bw* = 50, *roi* = 200 and *bw* = 100, *roi* = 100 and *bw* = 25, and *roi* = 200 and *bw* = 50. The first three combinations have $\frac{roi}{bw} = 2$ and the last two combinations have $\frac{roi}{bw} = 24$. The other parameters are *height* = 100 and target coverage = 95 percent. The number of tentative observers is 429025, 107584, or 26896 when *bw* = 25, 50, or 100. Each result is an average from 10 runs of the program. The

Table 2. Results for VIX Using 10, 50, or 250 Random Targets Per Point. Interval: the Interval between Successive Evaluation Points Along the Line Of Sight. Time: the Running Time of CUDA VIX in Seconds. RMSE: the RMSE of the Approximate Visibility Index Map. Observ.: the Number of Observers Selected for 95 Percent Coverage.

	10 targets			50 targets			250 targets		
Interval	Time	RMSE	Observ.	Time	RMSE	Observ.	Time	RMSE	Observ.
1	16.2	174.8	16323	80.4	172.3	15152	402.3	171.8	14833
2	9.3	175.8	16380	45.9	173.2	15184	230.1	172.7	14856
4	5.9	177.3	16436	28.1	174.8	15262	139.4	174.2	14908
8	4.1	180.6	16651	18.3	178.1	15444	89.6	177.6	15070
16	3.0	188.8	17254	12.7	186.6	15929	61.3	186.1	15469
32	2.3	209.9	18588	9.2	208.3	16921	43.8	207.9	16312
exp	2.9	187.5	16589	12.2	185.2	15267	58.5	184.8	14917
fib	3.4	183.8	16456	14.4	181.5	15197	69.9	181.0	14859
biexp	6.2	181.8	16506	29.3	179.4	15275	145.1	178.9	14924
bifib	7.5	179.1	16466	36.2	176.6	15215	179.7	176.1	14866

OpenMP program uses 50 threads with dynamic threads disabled. The results are the running time of VIX, FINDMAX, VIEWSHED, SITE, and the program (total) in seconds, and the number of observers. The total time includes I/O time, which is about 0.2 seconds. The number of observers is slightly different for the programs because of parallel execution and randomization. A smaller *bw* and more tentative observers produce longer time of VIEWSHED and SITE, but fewer observers. The time of VIX is roughly proportional to *log(roi)*. The time of FINDMAX is very small. The time of VIEWSHED is roughly proportional to *roi* and the number of tentative observers. The time of SITE varies a lot. It is related to *roi*, *bw*, and the number of observers. With a fixed *roi*, the time of SITE increases as *bw* decreases and the number of tentative observers increases. With a fixed *bw*, it may either decrease or increase as *roi* increases.

Table 4 shows the speedups of the parallel programs over the sequential program. The speedup of VIX is about 72 to 99 times for the CUDA program and 18 to 20 times for the OpenMP program. The speedup for the CUDA program decreases as *roi* increases, because evaluation points are farther from the observer and memory accesses for their elevations are less local, and because CUDA threads in a warp are instruction locked and access memory at the same time. The speedup for the OpenMP program also decreases a little as *roi* increases. The speedup of FINDMAX is about 5 to 11 times for the CUDA program and 13 to 15 times for the OpenMP program. The workload is too small for the CUDA program. As *bw* increases, the speedup for the CUDA program increases because each thread has more independent work, while the speedup for the OpenMP program decreases because the granularity of parallelism is larger. The speedup of VIEWSHED is about 39 to 59 times for the CUDA program and 17 to 18 times for the OpenMP program. An OpenMP program running on a 16-core Intel Xeon CPU can exhibit more than a factor of 16 speedup because each core can execute two hyperthreads.

For the same reason as for VIX, the speedup for the CUDA program decreases as *roi* increases (with the same *bw*), but increases as *bw* decreases and the number of tentative observers increases (with the same *roi*). However, the speedup for the OpenMP program does not decrease as *roi* increases (with the same *bw*). The speedup of SITE is about 7 to 31 times for the CUDA program and 3 to 8 times for the OpenMP program. The speedup increases as *bw* decreases (with the same *roi*) but may increase or decrease as *roi* increases (with the same *bw*). The speedup of the program is about 55 to 69 times for the CUDA program and 15 to 18 times for the OpenMP program. The speedup may increase or decrease as *roi* increases (with the same *bw*), and decreases as *bw* decreases (with the same *roi*). The CUDA program is about 3 to 4 times as fast as the OpenMP program.

Table 3. Results of the CUDA, OpenMP, and Sequential Programs, Averaged over 10 Runs. VIX, …, Total: Running Time in Seconds. Observers: the Number of Observers Selected for 95 Percent Coverage.

CUDA Implementation							
roi	*bw*	VIX	FIND.	VIEW.	SITE	Total	Observers
50	25	9.5	0.2	2.3	8.1	20.4	49199
100	50	11.8	0.1	2.8	2.3	17.2	15273
200	100	15.4	0.1	3.3	1.3	20.4	5461
100	25	11.8	0.2	10.1	4.2	26.5	14789
200	50	15.4	0.1	11.5	2.7	29.9	5185
OpenMP Implementation							
roi	*bw*	VIX	FIND.	VIEW.	SITE	Total	Observers
50	25	47.1	0.1	7.7	33.8	88.9	49201
100	50	53.2	0.1	7.6	9.1	70.2	15259
200	100	59.7	0.1	7.5	3.6	71.0	5449
100	25	52.9	0.1	30.0	12.3	95.5	14781
200	50	59.4	0.1	29.2	5.3	94.2	5181
Sequential Implementation							
roi	*bw*	VIX	FIND.	VIEW.	SITE	Total	Observers
50	25	939.8	1.5	138.0	249.2	1328.8	49263
100	50	1029.6	1.3	132.6	25.2	1188.9	15282
200	100	1100.2	1.1	128.7	9.3	1239.6	5452
100	25	1032.8	1.3	529.2	93.6	1657.2	14787
200	50	1109.9	1.2	509.2	35.3	1655.8	5186

Table 4. Speedups of the CUDA and OpenMP Programs Over the Sequential Program

CUDA implementation						
roi	*bw*	VIX	FIND.	VIEW.	SITE	Total
50	25	99.3	6.1	58.8	30.7	65.0
100	50	87.6	10.7	47.5	11.1	69.2
200	100	71.5	9.5	38.9	7.3	60.9
100	25	87.7	5.2	52.6	22.2	62.4
200	50	72.1	9.7	44.4	13.3	55.4
CUDA implementation						
roi	*bw*	VIX	FIND.	VIEW.	SITE	Total
50	25	20.0	15.3	17.9	7.4	14.9
100	50	19.4	14.2	17.4	2.8	16.9
200	100	18.4	13.0	17.1	2.6	17.5
100	25	19.5	13.8	17.6	7.6	17.3
200	50	18.7	13.9	17.4	6.6	17.6

Conclusions

We have optimized the multiple observer siting algorithm and parallelized it using CUDA on a GPU and using OpenMP on multi-core CPUs. A sample execution time is as follows. On a 16,385 × 16,385 terrain, assuming that observers can see out to a radius-of-interest of 100, finding the approximately 15,000 observers that have the most joint viewshed coverage takes 1,200 seconds for a sequential program. An OpenMP program with 16 threads takes 70 seconds, while a CUDA program on an NVIDIA K20Xm GPU takes only 17 seconds.

In general, the speedup of the CUDA program is about 50 to 70 times on an NVIDIA Tesla K20Xm GPU accelerator over the sequential program on a CPU core. The speedup of the OpenMP program is about 17 times on two Intel Xeon E5-2687W CPUs with 16 cores over the sequential program. Both techniques are very effective in accelerating the program. The CUDA program is faster, while the OpenMP program is easier to implement. Due to overhead, the GPU is more efficient for long computation, while the CPU is more efficient for short computation. If a program contains both long and short computations, it is possible to achieve greater efficiency by combining GPU and CPU parallel execution. There are two directions for future work. The first direction is to further increase speed by selecting multiple observers in each iteration of SITE to increase parallelism. Because parallel execution is fast, the second direction is to reduce the number of observers selected for a target coverage by computing a more accurate visibility index map or using more tentative observers.

Acknowledgments

This research was partially supported by CNPq, CAPES (Ciência sem Fronteiras Grant 9085-13-0), FAPEMIG and NSF Grant IIS-1117277.

References

Akbarzadeh, V., C. Gagné, M. Parizeau, M. Argany, and M.A. Mostafav, 2013. Probabilistic sensing model for sensor placement optimization based on line-of-sight coverage, *IEEE Transactions on Instrumentation and Measurement*, pp.293–303.

Axell, T. and M. Fridén, 2015. *Comparison between GPU and parallel CPU Optimizations in Viewshed Analysis*, Gothenburg, Sweden.

Bravo, J.C., T. Sarjakoski, and Jan Westerholm, 2015. Efficient implementation of a fast viewshed algorithm on SIMD architectures, *Proceedings of the 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Turku, Finland, pp.199–202.

Champion, D. C., J.E. Lavery, and Washington, D.C.,Department of the Army, 2002. Line of Sight in Natural Terrain Determined by L1-Spline and Conventional Methods, *Proceedings of the 23rd Army Science Conference*, Orlando, Florida.

Efrat, A., J.S. Mitchell, S. Sankararaman, and P. Myers, 2012. Efficient algorithms for pursuing moving evaders in terrains, *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*, Redondo Beach, California, pp.33–42.

Ferreira, C.R., M.V. Andrade, S.V. Magalhães, and W.R. Franklin, 2016. An efficient external memory algorithm for terrain viewshed computation, *ACM Transactions on Spatial Algorithms and Systems*.

Ferreira, C.R., M.V. Andrade, S.V. Magalhães, W.R. Franklin, and C.P. Guilherme C. Pena, 2014. A parallel algorithm for viewshed computation on grid terrains, *Journal of Information and Data Management*, pp.171–180.

Franklin, W.R., 2002. *Siting Observers on Terrain*, Heidelberg, Germany, Springer-Verlag, pp.109–120.

Franklin, W. R., and R. Clark, 1994. *Higher Isn't Necessarily Better: Visibility Algorithms and Experiments*, Bristol, Pennsylvania, Taylor & Francis, pp.751–770.

Franklin, W. R., and C. VOGT, 2004. Efficient observer siting on large terrain cells, *Proceedings of the Third International Conference on Geographic Information Science*, Adelphi, Maryland.

Franklin, W. R. and C. Vogt, 2006. *Tradeoffs when Multiple Observer Siting on Large Terrain Cells*, Heidelberg, Germany, Springer-Verlag, pp. 845–861.

Israelevitz, D. 2003. A fast algorithm for approximate viewshed computation, *Photogrammetric Engineering & Remote Sensing*, 79(6):767–774.

Kreveld, M.V., 1996. Variations on sweep algorithms: Efficient computation of extended viewsheds and class intervals, *Proceedings of the 7th International Symposium on Spatial Data Handling*, Delft, The Netherlands, pp.15–27.

Lee, J., 1992. Visibility dominance and topographic features on digital elevation models, *Proceedings of the 5th International Symposium on Spatial Data Handling*, Charleston, South Carolina, pp. 622–631.

Lindstrom, P., V. Pascucci, and the IEEE Computer Society, Washington, D.C., 2001. Visualization of large terrains made easy, *Conference Proceedings of the Conference on Visualization '01*, San Diego, California, pp. 363–371.

Line-Of-Sight Technical Group. 2004. U.S. Army Topographic Engineering Center, URL: *http://www.tec.army.mil/operations/programs/LOS/* (last date accessed:26 April 2017).

Luebke, D., M. Harris, J. Krüger, A.Lefohn, J. Owens, and J. Hensley, 2004. GPGPU: General purpose computation on graphics hardware, *ACM SIGGRAPH 2004 Course Notes*, Los Angeles, California.

Magalhães, S., V. Marcus, V. Andrade, and C. Ferreira, 2010. Heuristics to site observers in a terrain represented by a digital elevation matrix, *Proceedings of the XI Brazilian Symposium on Geoinformatics*, Campos do Jordão, Brazil, pp.110–121.

Magalhães, S., V. Marcus, V. Andrade, and W.R. Franklin, 2010. An optimization heuristic for siting observers in huge terrains stored in external memory, *Proceedings of the 10th International Conference on Hybrid Intelligent Systems*, pp.135–140.

Nagy, G., 1994. Terrain Visibility, *Computers & Graphics*, pp.763–773.

NVIDIA, 2016. *CUDA Parallel Computing Platform*, NVIDIA Corporation, Santa Clara, California.

Osterman, A., 2012. Implementation of the r.cuda.los module in the open source GRASS GIS by using parallel computation on the NVIDIA CUDA graphic cards, *Elektrotehniški Vestnik*, pp.19–24.

Osterman, A., L. Benedičič, and P. Ritoša, 2014. An IO-efficient parallel implementation of an R2 viewshed algorithm for large terrain maps on a CUDA GPU, *International Journal of Geographical Information Science*, pp.2304–2327.

Pena, G.C., S.V. Magalhães, M.V. Andrade, W.R. Franklin, C.R. Ferreira, and W. Li, 2014. An efficient GPU multiple-observer siting method based on sparse-matrix Multiplication, *Proceedings of the 3rd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*, Dallas, Texas, pp.54–63.

Rana, S., 2003. Fast approximation of visibility dominance using topographic features as targets and the associated uncertainty, *Photogrammetric Engineering & Remote Sensing*, 79(7):881–888.

Strnad, D., 2011. Parallel terrain visibility calculation on the graphics processing unit, *Concurrency and Computation: Practice and Experience*, pp.2452–2462.

Wang, J., G.J. Robinson, and Kevin White, 2000. Generating viewsheds without using sightlines, *Photogrammetric Engineering & Remote Sensing*, pp.87-90.

Zhao, Y., A. Padmanabhan, and Shaowen Wang, 2013. A parallel computing approach to viewshed analysis of large terrain data using graphics processing units, *International Journal of Geographical Information Science*, pp. 363–384

Received 30 July 2016; accepted 16 January 2017; final version 15 April 2017)