An efficient GPU multiple-observer siting method based on sparse-matrix multiplication

Guilherme C. Pena
Universidade Fed. de Viçosa
Viçosa, MG, Brazil
guilherme.pena@ufv.br

Salles V. G. Magalhães
Rensselaer Polytechnic Inst.
Troy, NY, USA
salles@ufv.br

Marcus V. A. Andrade
Universidade Fed. de Viçosa
Viçosa, MG, Brazil
marcus@ufv.br

W. Randolph Franklin
Rensselaer Polytechnic Inst.
Troy, NY, USA
mail@wrfranklin.org

Chaulio R. Ferreira
Universidade Fed. de Viçosa
Viçosa, MG, Brazil
chaulio.ferreira@ufv.br

Wenli Li
Rensselaer Polytechnic Inst.
Troy, NY, USA
liw9@rpi.edu

ABSTRACT

This paper proposes an efficient parallel heuristic for siting observers on raster terrains. More specifically, the goal is to choose the smallest set of points on a terrain such that observers located in these points are able to visualize at least a given percentage of the terrain. This problem is NP-Hard and has several applications such as determining the best places to position (site) communication or monitoring towers on a terrain. Since siting observers is a massive operation, its solution requires a huge amount of processing time even to obtain an approximate solution using a heuristic. This is still more evident when processing high resolution terrains that have become available due to modern data acquiring technologies such as LIDAR and IFSAR.

Our new implementation uses dynamic programming and CUDA to accelerate the swap local search heuristic, which was proposed in previous works. Also, to efficiently use the parallel computing resources of GPUs, we adapted some techniques previously developed for sparse-dense matrix multiplication.

We compared this new method with previous parallel implementations and the new method is much more efficient than the previous ones. It can process much larger terrains (the older methods are restrictive about terrain size) and it is faster.

Categories and Subject Descriptors

F.2.2 [**Nonnumerical Algorithms and Problems**]: Geometrical problems and computations

General Terms

Algorithms, Experimentation, Performance

3rd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data (BigSpatial) 2014 Dallas/Fort Worth, TX, USA

Keywords

Siting, Terrain Visibility, Viewshed, GPU parallel algorithm.

1. INTRODUCTION

Terrain modeling plays an important role in geographic information science (GIS). Many applications concern visibility, that is, determining the set of points that are visible from a particular point, called the observer, which can be located at some height above the terrain. These applications include telecommunications, environmental planning, autonomous vehicle navigation and military monitoring [1, 6, 7, 12]. Among these applications, we can point out the siting problem, where the goal is to select a set of observers in order to "optimally cover the terrain". These observers may represent radio, TV, Internet or mobile phone towers, or monitoring cameras or towers [2, 3].

As described in [12], the siting problem is NP-Hard and, therefore, there is no known efficient algorithm to find its optimal solution. Thus, in general, a heuristic is used to obtain an approximate solution. But even obtaining approximate solutions for this optimization problem can demand a long processing time since sometimes it is necessary to process a huge amount of high-resolution geographic data. For example, new satellite sensors are able to sample the Earth surface elevations at 1m resolution generating very huge elevation matrices.

Thus, the geographic information system applications have required the development of some advanced techniques to process this volume of data. A technique that has been successfully used is to design parallel algorithms based on general purpose graphics processing units (GPGPUs), which are present in most current graphics cards.

This paper presents an approximate solution for an instance of the multiple observer siting problem where the goal is to determine a set of observers on a terrain represented by an elevation matrix such that these observers together can achieve a given visual coverage of the terrain. A first solution for this problem was presented in [5], which was based on a greedy strategy. In this paper, we extend that method including a local search heuristic based on a swapping strategy to achieve the desired coverage using a smaller number

of observers. As the main contribution of this paper, this heuristic was implemented in parallel using Graphics Processing Units (GPUs) and dynamic programming.

This local search strategy to reduce the number of observers was already used in [10,15] and, as shown in those papers, it allows a reduction of up to 20% in the number of observers required to achieve the desired coverage (which may represent an important improvement since the observer can be an expensive facility as, for example, a communication tower).

Both methods are very restricted since they can not process large terrains. As presented in those papers, the largest terrains that these methods can process have about 3601 × 3601 cells. However, applications usually have to process much larger terrains. Thus, in this paper, we present an efficient new implementation, named *SparseSite*, which is able to process larger terrains (we tested it on terrains with up to 2×10^8 cells). Also, while processing larger terrains, this new method is, very often (mainly in larger terrains), faster than the other methods (we also executed the new method on small terrains to compare its execution time and memory usage with the other methods).

2. TERRAIN VISIBILITY DEFINITIONS

A *terrain* represents a region of the earth surface where the terrain's value at any point is the elevation of the corresponding earth surface point above a reference ellipsoid called the *geoid* that represents sea-level. For this paper, a terrain is represented by a matrix of elevation posts on a square grid, whose vertical and horizontal spacing is uniform either in distance, e.g., 10m, or in angle, e.g. 1 arc-second.

An *observer* is a point in the space that wants to see or communicate with other points in the space, called *targets*. The notations for observer and target are O and T. The *base points* of O and T are the points on the geoid directly below O and T respectively, which are denoted as O_b and T_b. Both O and T are at height $h \geq 0$ above O_b and T_b. All symbols used in this work are shown in Table 5.

The *radius of interest*, R, of O is the radius of the circle centered on O_b that contains all points that can be seen by the observer in the absence of obstructions. E.g., if O is a radio transmitter, R is a function of the transmitter power and receiver sensitivity. For convenience, R is usually compared to the distance between O_b and T_b rather than between O and T, which is equivalent when h is much smaller than the radius of the earth.

A target T is visible from O iff $|T_b - O_b| \leq R$ and there is no terrain point blocking the line segment, called the *Line of Sight (LOS)*, between O and T; see Figure 1. In this figure, T_1 is visible from O but T_2 is not.

The *viewshed*, V, of O is the set of base points whose corresponding targets are visible from O. In general, V is stored as a bit matrix of size $2R \times 2R$ where 0 represents a non-visible point and 1 represents a visible point.

The *visibility index*, ω, of O is the number of targets that are visible from O. Points with a large ω are usually good candidate places to site observers in order to maximize the

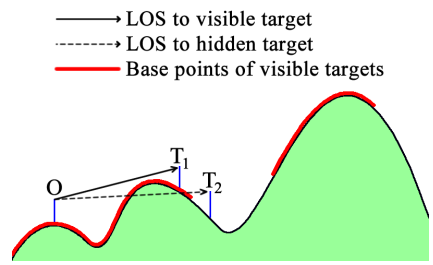

Figure 1: Visibility queries using a line of sight.

area of the terrain that is seen by at least one observer [6].

The *joint viewshed*, $\mathcal{V}$, of a set of observers $\mathcal{S} = \{O_i\}$ is the union of the individual viewsheds V_i, i.e., the bitwise-*or* of their bit matrices.

The *joint visibility index*, Ω, of $\mathcal{S}$ is the number of targets in the terrain that are visible from at least one observer in $\mathcal{S}$. Usually, Ω is normalized as a percentage of the terrain area.

Multi-observer siting means optimizing the locations of a set of observers such that Ω is as large as possible. This is an NP-Hard problem [12] and has important practical facilities-location applications, such as siting mobile phone towers, fire monitoring towers, and radar systems.

In this paper we will consider the following equivalent multi-observer siting problem: to obtain a set of observers whose joint visibility index Ω is, at least, a given percentage of the terrrain.

3. RELATED WORK

There are some important related works addressing the siting problem. Ben-Moshe [2] presents an algorithm to site facilities that considers radio coverage, frequency allocation and connectivity. The input includes a weighted set of demand locations, a set of feasible facility locations and a distance function that measures the cost of travel between a pair of locations. Ben Shimol et al [3] describe an algorithm to site a minimal set of fixed-access relay antennas on a given terrain to generate the communication links between multiple base stations. Although the goals of these two methods are a little different of the problem addressed on this paper, they use some important concepts related to our approach.

Considering that the observer siting problem is NP-Hard, Franklin [5] proposed an approximate heuristic solution, called *Site*, to find a set of observers to cover the terrain. This method uses a greedy approach to obtain a set $\mathcal{S}$ of observers such that a given percentage of the terrain is covered. Shortly, the solution $\mathcal{S}$ is initialized as empty and a set $\mathcal{P} = \{P_i\}$ of candidate observers is selected. Then, at each step, the P_i that will most increase the current joint visibility index of $\mathcal{S}$ is inserted into $\mathcal{S}$.

Other papers describing solutions to site observers on terrains are [9–11,15]. In [11] the method *Site* was extended to process huge terrains stored in external memory, where the main idea is to subdivide the terrain in smaller pieces (subregions) and process each piece in the internal memory. In

order to consider the influence of observers sited near to the borders of the subregions, each subregion is augmented with a band of width R (the observer radius of interest) around it. Also, the viewshed representation used in the method *Site* was improved to require a smaller amount of memory.

Since an observer may represent an expensive facility (for example, a telecommunication tower), it is important to develop methods that can achieve a same coverage using fewer observers than the greedy strategy implemented in the method *Site*. This goal is addressed by the three methods *Site+, SiteGPU* and *SiteGSM*, respectively described in [9, 10, 15], where the idea is to try to reduce the number of observers selected by the original method *Site*. Basically, these methods extend the method *Site* including, in each iteration of the greedy algorithm, a *swap* local search to try to improve the visibility index of the current (partial) solution while keeping fixed the number of observers used. Thus, the greedy algorithm may achieve a feasible solution in fewer iterations (that is, using less observers).

These three methods differ from each other mainly in how the local search is implemented. Magalhães et al [9] implemented the local search using CPU sequential programmings and, in [10], they presented a first parallel implementation of the local search using CUDA. As the amount of data being processed is too big, these data are stored and processed in the GPU's global memory (the biggest, but the slowest memory in the GPU's memory hierarchy). In [15] the parallel routines were reimplemented to use a faster memory access pattern and to store (temporally) the data being processed in the GPU's shared memory which is much faster than the global memory. These improvements were based on techniques used in fast matrix multiplication algorithms for GPUs since the viewshed overlays adopt a data access pattern similar to matrix multiplication. A similar strategy was used in [4] where the Floyd-Warshall shortest path algorithm was adapted to be executed using GPU threads. Also, the heuristic efficiency was improved using dynamic programming.

It is important to mention that all three previous methods (*Site+, SiteGPU* and *SiteGSM*) and also the method proposed in this paper always obtain a same solution, that is, select the same set of observers because they are based on a same heuristic. But, although the two methods *SiteGPU* and *SiteGSM* are efficient (considering their execution time)[1], they are very restricted since they require too much memory and, the GPUs usually have a small amount of memory (ranging from 1 GB for low-end GPUs to 6GB for high-end scientific GPU accelerators). Thus, they are not suitable for recent GIS applications that demand the processing of high resolution terrains. In this paper, we propose a more efficient parallel implementation for observer siting on terrains. This new method, named *SparseSite*, not only is able to site observers faster than *SiteGPU* and *SiteGSM*, but mainly uses less memory, allowing the processing of much larger terrains. As the tests showed, the improvements are particularly important because observer siting is a very massive application that could take several hours (or even days) of processing time when performed using a sequential approach. Also, the lower use of memory allows the method to be used even in inexpensive low-end GPUs

[1]In fact, as described in [15], *SiteGSM* is more than 20 times faster than *SiteGPU*.

4. THE PROPOSED METHOD

We propose a new method for siting observers on terrains, named *SparseSite*, which is an important improvement of the method *SiteGSM* described in [15]. In this new method, the observer siting heuristic is the same heuristic used by *SiteGPU* and *SiteGSM*, that is, observers are sited using a greedy algorithm improved with the *swap* local search. But, as described in details later, this new implementation circumvents an important restriction in the previous methods since it can process (much) larger terrains. It is important mentioning that, despite we used a greedy method with the *swap* local search, our fast implementation of the *swap* can be used to accelerate other observer siting heuristics once the local search is often a bottleneck of other heuristics such as Simulating Annealing or GRASP.

4.1 The Local Search - Swap

Given an initial solution S, the objective of the local search is to iteratively perform small changes in S such that each change transforms S in a solution S' with the same number of observers as S, but with higher joint visibility index. In the swap local search strategy, each small change corresponds to swapping an observer in S with an observer not in S. More specifically, given a solution S and a set with n candidate observers whose viewsheds are represented by $V = \{V_1, \cdots, V_n\}$, the local search iteratively changes S by replacing the observer $V_i \in S$ with the observer $V_j \in V$, where (V_i, V_j) is the pair of observers that maximizes the visibility index of $S - V_i + V_j$. The swap local search ends when no swap of observers leads to a better solution.

In each iteration of the local search it is necessary to evaluate all solutions S' (called neighbors) that can be created by replacing one observer in S with an observer not in S. Figure 2 presents an example of the solutions that are evaluated performing one iteration of the swap local search in the solution $S = \{V_1, V_2, V_3\}$.

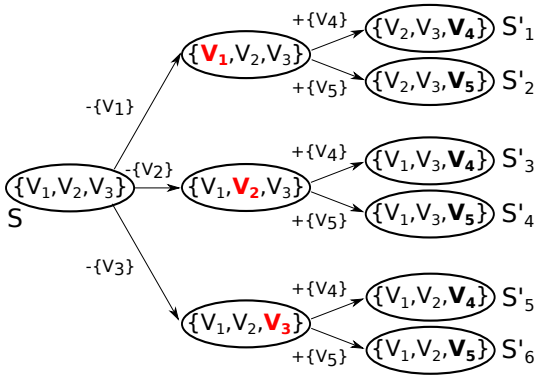

Figure 2: Given $V = \{V_1, V_2, V_3, V_4, V_5\}$, each S'_i is obtained performing a swap in $S = \{V_1, V_2, V_3\}$.

To simplify the notation, a solution $S = \{V_{i_1}, \cdots, V_{i_k}\}$ will be written as $S = \{i_1, \cdots, i_k\}$ indicating that the solution, in fact, corresponds to the joint viewshed of the observers whose indices are $i_1, \cdots, i_k$. Thus, the local search may

be implemented (sequentially) as follows: given the set of candidate viewsheds $V = \{V_1, \cdots, V_n\}$, let S be a solution composed of k viewsheds, i.e., $S = \{i_1, \cdots, i_k\}$ such that the joint viewshed of S is $V_{i_1} \oplus \cdots \oplus V_{i_k}$, where $\oplus$ represents the union operation between two viewsheds.

Furthermore, let $\mathcal{V}_{\overline{r}}$ be the joint viewshed of all viewsheds in S except V_{i_r} and assuming the viewsheds and joint viewsheds are linearized using a row-major order, let $\mathcal{V}$ be the matrix storing all $\mathcal{V}_{\overline{r}}$ for $r = 1, \cdots k$. In each iteration, the neighbors of S are generated in order to find the best solution for the next iteration. The visibility indices of the neighbor solutions are obtained computing the number of visible points in $V_j \oplus \mathcal{V}_{\overline{r}}$ for $r = 1 \cdots k$ and $j = 1 \cdots n$, with $j \neq r$.

The most time-consuming step in this heuristic is computing the joint visibility index for each neighbor solution. Algorithm 1 presents the code for this step that computes the number of visible points in $V_j \oplus \mathcal{V}_{\overline{r}}$ (for all $r = 1 \cdots k$ and $j = 1 \cdots n; j \neq r$) and stores them in the element $Vix[j][r]$. In the next step, this matrix will be used to find the best neighbor of S.

Algorithm 1 Calculate the Vix matrix where *vsize* is the number of points in each viewshed, k is the number of observers in the solution S and n is the number of candidate observers. The output is the matrix Vix, where $Vix[j][r]$ is the joint visibility index of a solution replacing observer r with j.

```
1:  Vix[n][k] ← {{0}}
2:  V[k][vsize] ← {{0}}
3:  for r ← 1 to k do
4:     for m ← 1 to k do
5:        if r ≠ m then
6:           s ← S[m]
7:           for w ← 1 to VSize do
8:              V[r][w] ← V[r][w] or V[s][w]
9:           end for
10:       end if
11:    end for
12: end for
13: for r ← 1 to k do
14:    for j ← 1 to n do
15:       for w ← 1 to vsize do
16:          Vix[j][r] ← Vix[j][r] + (V[j][w] or V[r][w] )
17:       end for
18:    end for
19: end for
20: return Vix
```

For efficiency, in this work the viewsheds are packed in 64-bit words (where each word represents the visibility of 64 points). Thus, the viewshed unions and visibility indices can be computed using, respectively, bitwise-or operator and bit population count functions, which are available in the hardware of most current computers.

4.2 An Efficient Swap Heuristic Implementation

Observe that Algorithm 1 performs $\Theta(k^2)$ union operations $\oplus$ to compute $\mathcal{V}$. This number of union operations can be reduced using dynamic programming as described below.

Given a solution $S = \{i_1, \cdots, i_k\}$, for each $r \in \{1, \cdots, k\}$, we have $\mathcal{V}_{\overline{r}} = (V_{i_1} \oplus \cdots \oplus V_{i_{r-1}}) \oplus (V_{i_{r+1}} \oplus \cdots \oplus V_{i_k})$. Let λ_r^- and λ_r^+ be

$$\lambda_r^- = V_{i_1} \oplus \cdots \oplus V_{i_{r-1}}$$
$$\lambda_r^+ = V_{i_{r+1}} \oplus \cdots \oplus V_{i_k}$$

Notice that both λ_r^- and λ_r^+ can be obtained by the following recurrence relations:

$$\lambda_1^- = \emptyset \text{ and } \lambda_r^- = \lambda_{r-1}^- \oplus V_{i_{r-1}} \text{for all } r \in 2, \cdots, k$$
$$\lambda_k^+ = \emptyset \text{ and } \lambda_r^+ = V_{i_{r+1}} \oplus \lambda_{r+1}^+ \text{for all } r \in 1, \cdots, k-1$$

For example, Figure 3 illustrates the $\mathcal{V}$ computation for $k = 5$: in this figure, each row r of matrix $\mathcal{V}$ contains the joint viewsheds $\mathcal{V}_{\overline{r}}$. Observe that the value of $\mathcal{V}_{\overline{r}}$ may be obtained by performing the union between the corresponding λ_r^- and λ_r^+. Also, each row r in λ^- can be computed from the previous row $r-1$ using only one $\oplus$ operation. Analogously, the λ^+ values can be obtained in a similar way using the reverse order.

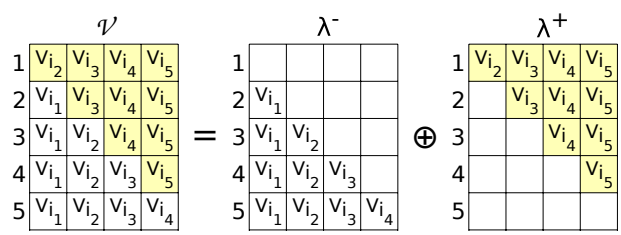

Figure 3: Matrix illustrating $\mathcal{V}_{\overline{r}}$ in a solution with $k = 5$ observers.

Based on these recurrence relations, Algorithm 2 uses dynamic programming to compute a matrix $\mathcal{V}$ that stores $\mathcal{V}_{\overline{r}}$, for $r = 1 \cdots k$. Notice that this algorithm performs only $\Theta(k)$ viewshed unions and can replace the piece of code composed by lines 2 to 11 in Algorithm 1 where are performed $\Theta(k^2)$ viewshed unions.

As presented in the *SiteGSM* algorithm [15], the Vix computation given in Algorithm 2 can be implemented in the GPU using the following strategy. The viewsheds were kept in GPU's global memory. Then, the union of viewsheds (loop in lines 4, 11 and 17 of Algorithm 2) were performed using GPU's threads, that is, each thread performs a bitwise-or operation with one element of a viewshed and the corresponding element of another viewshed.

After computing $\mathcal{V}$, the next step is to compute the joint visibility index of the neighbor solutions, as performed by lines 12 to 18 in Algorithm 1. A straightforward implementation of this step in GPU was presented in [10], where all viewsheds are kept in the GPU's global memory and, then, each element of matrix Vix (that stores the joint visibility index) is computed using a parallel algorithm to overlap a pair of viewsheds followed by a parallel reduction operation to determine the number of visible points. However, this strategy does not take advantage of the GPU resources efficiently because it requires too many accesses to the global

Algorithm 2 Compute the matrix $\mathcal{V}$ that stores $\mathcal{V}_{\overline{r}}$, for $r = 1 \cdots k$ using a dynamic programming strategy.

```
 1: V1[k][vsize] ← {{0}}
 2: for r ← 2 to k do
 3:    lf ← S[r − 1]
 4:    for w ← 1 to vsize do    // using GPU threads
 5:       V1[r][w] ← V1[r − 1][w]  or  V[lf][w]
 6:    end for
 7: end for
 8: V2[k][vsize] ← {{0}}
 9: for r ← k − 1 to 1 do
10:    rg ← S[r + 1]
11:    for w ← 1 to vsize do    // using GPU threads
12:       V2[r][w] ← V2[r + 1][w]  or  V[rg][w]
13:    end for
14: end for
15: V[k][vsize] ← {{0}}
16: for r ← 1 to k do
17:    for w ← 1 to vsize do    // using GPU threads
18:       V[r][w] ← V1[r][w]  or  V2[r][w]
19:    end for
20: end for
```

memory, which is much slower than other memories such as the shared memory.

In order to make a better use of the GPU memory hierarchy, also in *SiteGSM* [15], we proposed a strategy based on a fast GPU matrix multiplication algorithm. Notice that each element of the joint visibility index (that is, each position of the Vix matrix) is obtained overlapping one row of matrix V with another row of matrix $\mathcal{V}$ using a bitwise-or operation (see lines 12 to 18 of Algorithm 1). It means that the two matrices are swept in a row major order, but the matrices could be reorganized such that the overlapping could be computed using an access pattern similar to matrix multiplication. More precisely, line 15 in Algorithm 1 can be replaced with

$$Vix[j][r] \leftarrow Vix[j][r] + V[j][w] \ \textbf{ or } \ \mathcal{V}^T[w][r]$$

where $\mathcal{V}^T$ is the transposed matrix of $\mathcal{V}$.

Thus, the Vix matrix can be computed using a simple adaptation of some very fast algorithm for matrix multiplication in GPU. In particular, we adapted the algorithm presented in [14], replacing the multiplication operation by a bitwise-or followed by a binary population count operation. This algorithm subdivides the matrices into blocks, which are loaded iteratively in the GPU's shared memory as the multiplication process is performed. Therefore, most of the algorithm accesses are to the shared memory which is much faster than the global memory.

4.3 Using a sparse-dense matrix multiplication algorithm

Since the viewsheds usually contain a large amount of non visible points (mainly when the radius of interest is much smaller than the dimensions of the terrain), the swap heuristic performance can be improved storing the matrix V (containing the candidate observers viewsheds[2]) using a sparse matrix format and, then, using an algorithm to overlay the sparse viewsheds in V with the joint viewsheds in $\mathcal{V}$. The matrix $\mathcal{V}$ is stored as a dense matrix since each of its lines is composed by a joint viewshed which is usually much denser than an individual viewshed.

In this work, the matrix V was coded using the ELLPACK-R matrix format [8]. The overlay of the viewsheds in V with the joint viewsheds in $\mathcal{V}$ were performed using an adaptation of the sparse-dense matrix multiplication algorithm proposed by [8].

Observe that in a (conventional) matrix multiplication, each entry (i, j) in the resulting matrix corresponds to the scalar product of the i-th row of the first matrix with the j-th column of the second matrix. Thus, when replacing the multiplication operation in the scalar product with a bitwise *or* followed by a population count operation, the resulting value corresponds to the visible area of the union of the viewshed i with the joint viewshed j.

However, since the sparse-dense matrix multiplication algorithm proposed in [8] does not process 0 in the first (sparse) matrix then this straightforward adaptation does not work in this case because while in the multiplication a 0 operand yields 0 as result, this is not true for the *or* operation.

To circumvent this problem, we changed the Vix matrix computation as following: instead of evaluating the total visible area of the overlay between each viewshed in V with each joint viewshed in $\mathcal{V}$ to select that one with the largest visible area, we compute the area increment[3] that each viewshed in V would generate to the joint viewsheds in $\mathcal{V}$ and, at the end, the entries in the Vix matrix are obtained by adding the corresponding increment to the visible area of each joint viewshed. Notice that if an entry has value 0 in V, this entry will never increment any entry in $\mathcal{V}$ and, therefore, this 0 entry in V does not need to be processed. Thus, the sparse-dense matrix multiplication algorithm may be adapted to efficiently compute the contribution area.

In other words, we use the same access pattern of the sparse-dense multiplication algorithm to overlap the matrices V and $\mathcal{V}$ replacing the multiplication operation $a \times b$ (where a is an element of V and b is an element of $\mathcal{V}$) with the operation $((a \textbf{ or } b) \textbf{ and } \sim a)$ followed by a bitwise population count. Notice that, when a viewshed b is overlapped with a, the operation $((a \textbf{ or } b) \textbf{ and } \sim a)$ returns the points that are visible in b but not in a. Thus, this operation returns the contribution (exclusively) from b to the visible area, that is, the area increment.

4.4 Reducing the swap heuristic's memory usage

As previously described, given a partial solution S with k observers and a set of n candidate observers V, the proposed

[2]Each viewshed is stored using a linear row-major order array.

[3]The area increment represents how much the area of a joint viewshed would be incremented if this joint viewshed is overlapped with that viewshed

algorithm tries to improve the visual coverage of the (partial solution with) k observers performing a swap heuristic to check if there is a pair of observers (p_i, p_j) with $p_i \in S$ and $p_j \in V$ such that the visual coverage of S would increase by replacing p_i with p_j. This process checks all p_i with $i = 1 \cdots k$ and all p_j with $j = 1 \cdots n$. In this paper, we present an implementation for this strategy that is not only time efficient, but also that can reduce the amount of memory used by the matrix that stores the candidate observers viewsheds, since these viewsheds are stored using a sparse matrix format. The $\mathcal{V}$ matrix is still stored as a dense matrix because it stores the joint viewsheds that are "dense" (that is, they have, in general, many visible points). But, even using this representation, often it is not possible to keep the whole $\mathcal{V}$ matrix in the GPU memory (as required by that method) because the memory of a GPU is usually much smaller than the CPU main memory (in general the GPU memory size ranges from 1 GB for a gaming GPU to 6GB for high-end scientic GPUs such as a nVIDIA K20x).

Thus, to allow processing larger terrains, we proposed a new strategy where the basic idea is to split the $\mathcal{V}$ and Vix matrices computation in small parts composed by some rows of the whole matrix. More precisely, the idea is to iteratively compute a submatrix $\mathcal{V}_{\overline{i \cdots j}}$ composed by the rows $i, i+1, \cdots, j$ of $\mathcal{V}$ and, then, to obtain the submatrix $Vix_{i,j}$ corresponding to the visible area of the overlay of each viewshed in V with each joint viewshed $\mathcal{V}_{\overline{i}} \cdots \mathcal{V}_{\overline{j}}$. These overlays are still computed as described in Section 4.3. That is, using an adaptation of sparse-dense matrix multiplication algorithm to overlay the viewsheds in the sparse matrix formed by the candidate observers with a dense matrix where each row represents a joint viewshed. After concluding each step, the resulting submatrix $Vix_{i,j}$ is copied to the CPU's memory and the corresponding entries of this submatrix are used to fill the Vix matrix .

Notice that the dynamic programming approach presented in Section 4.2 needs to be adapted since, to compute $\mathcal{V}_{\overline{k}}$, it is necessary to have λ^-_{k-1} and λ^+_{k+1} previously computed. As the computation of the submatrix $\mathcal{V}_{\overline{i \cdots j}}$ starts in row i and ends in row j, the base case for λ^- must be λ^-_i and the base case for λ^+ must be λ^+_j. See an example in Figure 4. Thus, before starting the computation of $\mathcal{V}_{\overline{i \cdots j}}$ the matrix λ^-_i is initialized with the union of viewsheds of points $S_1, S_2, \cdots, S_{i-1}$ and λ^+_j is initialized with the union of the viewsheds of points $S_j, S_{j+2}, \cdots, S_k$.

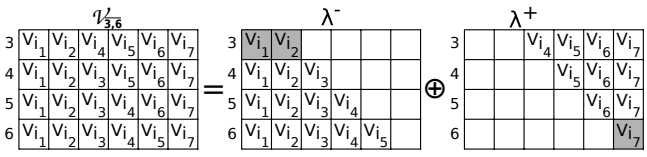

Figure 4: Matrix illustrating $\mathcal{V}_{\overline{3 \cdots 6}}$ in a solution with $k = 7$ observers. Notice that the base case for computing matrix λ^- is λ^-_3 (indicated in gray) and, similarly, the base case for computing λ^+ is λ^+_6.

The time complexity to process each slice of size n_r of the matrix $\mathcal{V}$ (that is, each submatrix with n_r rows of the matrix $\mathcal{V}$) is $\Theta(n_r) + \Theta(k - n_r)$, since to process each slice $\mathcal{V}_{\overline{i \cdots j}}$ it is necessary the initialization of λ^-_i (this process performs $i-1$ viewshed overlays) and λ^+_j (that performs $k-j$ overlays) resulting in $k - j + i - 1 = k - (j - i + 1) = k - n_r$ viewshed overlays. After initializing the base cases, the computation of each line of the submatrix requires 1 viewshed overlay. In total, n_r overlays are performed. Since the matrix $\mathcal{V}$ is subdivided into $\Theta(\frac{k}{n_r})$ slices, the total processing time of the dynamic programming algorithm is $\Theta(\frac{k}{n_r}) \times (\Theta(n_r) + \Theta(k - n_r)) = \Theta(k) + \Theta(\frac{k^2}{n_r}) = \Theta(\frac{k^2}{n_r})$.

Thus, the larger the size of $\mathcal{V}$ matrix that is kept in the GPU's memory, the more efficient the computation is. When n_r is close to k, the efficiency of this new approach is similar to the efficiency of the dynamic programming algorithm presented in Section 4.2, that is, $\Theta(\frac{k^2}{n_r}) \approx \Theta(\frac{k^2}{k}) = \Theta(k)$. On the other hand, if n_r is too small, this efficiency will be closer to $\Theta(k^2)$. Therefore, n_r should be chosen such that the submatrices use the maximum amount of memory available in the GPU.

5. EXPERIMENTAL RESULTS

In order to evaluate the proposed method, we defined a set of tests and compared it against *Site+* [9], that is a sequential implementation of the *swap* heuristic and against *SiteGSM* [15], which, as far as we know, is the fastest parallel implementation of the *swap*. It is important to mention that we do not compare the proposed method against traditional *GISs* such as GRASS and ARCGIS because, to the best of our knowledge, these softwares provide tools to perform visibility analysis (for example, to compute viewsheds), but they do not contain any method that is able to perform observer siting. Also, the method proposed in [10] was not evaluated in our tests because, as shown in [15], *SiteGSM* is much faster than it.

All heuristics were implemented in C++/CUDA and compiled using *g++* 4.8 (for the sequential heuristic) and *nvcc* 4.0 (for the parallel heuristics) with optimization level (-O3). The tests were executed on a computer running Ubuntu 12.04 LTS Linux and with the following hardware configuration: Dual Intel Xeon E5-2687 3.1GHz processor, 128GiB of memory and GPU NVidia Tesla Kepler K20x with 6GiB of global memory, 48KB of shared memory per block and 2688 CUDA processing cores running CUDA 5.0.

The tests were performed on 4 terrains, that contain 30-meter-resolution elevation data from the state of Illinois, in the United States, and were obtained from the NASA SRTM project [13]. The terrains, that are numbered from 1 to 4, represent square regions and contains, respectively, the following number of cells: 1201^2, 3601^2, 7500^2 and 15000^2.

Initially, the set $\mathcal{P}$ of candidate observers was generated using the observer selection step of the *Site* method [5], that selects observers with high visibility indices in the terrain. The size of the candidate points set for the terrains $1, 2, 3$ and 4 were, respectively, 1008, 3008, 5625 and 22500. Then, the viewshed of each candidate observer was computed using the viewshed utility also available in the *Site* method. During the viewshed computation, the observer height above the terrain was defined as 30 meters, since this value represents typical elevation for communication antennas.

Table 1: Processing time and memory usage of *SparseSite* in large terrains.

Ter.	R	Ω	#Obs.	*SparseSite*	
				Time(sec)	Memory(MB)
3	200	75%	346	720	1043
		85%	410	1158	
		95%	517	1950	
	400	75%	87	279	1590
		85%	102	381	
		95%	126	610	
4	400	75%	354	11830	3819
		85%	420	19011	
		95%	549	33863	

Table 2: Processing-time and memory usage of *SparseSite* as a function of the number of rows (n_r) of the dense matrix kept in the memory. These tests were performed on Terrain 3, using radius of interest equal to 200 cells and target coverage of 95%.

	SparseSite	
n_r	Time(sec)	Memory(MB)
1	4533	217
5	2069	304
10	2043	410
20	1939	621
40	1952	1043
80	1958	1887
160	1957	3577
260	1972	5688

Since the biggest contribution of *SparseSite* is its ability to efficiently process high resolution terrains, in our first set of tests we evaluated its efficiency during the processing of the bigger terrains. In these tests, it was not possible to compare *SparseSite* against other methods since *SiteGSM* was not able to handle these large terrains in the GPU memory and *Site+* was not able to terminate its execution even after running for 5 days.

As described in Section 4.4, *SparseSite* can be configured to use different sizes for the portion of the sparse matrix that is kept in the GPU's memory. Thus, in exchange for a bigger processing time, the heuristic may be configured to process large terrains even in GPUs that do not have enough memory to store all the data being processed. To evaluate how *SparseSite* behaves as the amount of memory available to the heuristic is reduced, we performed a set of tests where the parameter n_r (the parameter that defines the number of rows of the sparse matrix kept in the GPU's memory) varies. Table 2 presents the running-time (in seconds) and total memory usage (in MB) of *SparseSite* as a function of n_r. These tests where performed in Terrain 3 using radius of interest with 200 cells and target coverage of 95%.

Observe that, except in the case where only one line of the sparse matrix is kept in the memory, there is little variation in the performance of the heuristic. In fact, the difference between the test case where only 5 lines were kept in the memory and the test case where n_r equals to 260 (this was the largest value for n_r that we managed to use in the K20x GPU, that has 6GB of memory) is smaller than 5%. This result is important because it indicates that, differently of the previous methods, the proposed heuristic could process big terrains using an amount of memory that is available even in lower ends GPUs. In fact, as it will be explained latter, *SiteGSM* was not able to process the terrain evaluated in this test, even though, *SparseSite* managed to efficiently process it using only 304 MB of memory.

To compare *SparseSite* against other methods, we performed a set of tests on smaller terrains that can be processed in a feasible amount of time by the sequential implementation of the swap heuristic (*Site+* [9]) and, also, that can be processed using less than the available 6GB of memory by the previous parallel implementation (*SiteGSM* [15]). The test scenarios use radii of interest 100, 200 and 300 cells for the smallest terrain, 200, 300 and 400 cells for the largest one. For each terrain, the desired coverages (joint visibility indices) were 75%, 85% and 95% of the terrain area. The results are presented in Table 3. Column #Obs shows the number of observers sited in each case and the speedup of each method (if compared against the sequential method) is presented in parenthesis.

In all situations both *SparseSite* and *SiteGSM* were much faster than *Site+*. Notice that the greatest speedup obtained by *SparseSite* was 7352×, while the greatest speedup obtained by *SiteGSM* was 2762×. Observe that in most of the cases *SparseSite* was faster than *SiteGSM*, being up to 2.7 times faster. However, in the cases where the radius of interest leads to viewsheds whith size that represents a considerable portion of the terrain, *SparseSite* was faster. This behaviour is explained because in these cases the matrices being processed are less sparse and, also, in these cases the number of observers needed to achieve the target coverage is very small (for example, in the test case in Terrain 1 with radius of interest equal to 300 cells and target coverage of 75%, the algorithm used only 4 observers). Thus, the overhead of the method that was specially designed for sparse matrices reduces its performance.

As shown by Magalhães et al. [9], the use of local search procedures with greedy heuristics can generate solutions that use fewer observers than the solutions obtained by methods that use a pure greedy method. Even though the focus of this paper is to present a fast parallel implementation for the *swap* local search and evaluate its performance, in the last set of experiments we evaluated the quality of the solutions obtained by the proposed method. Table 4 presents a comparison between *SparseSite* (the greedy method improved with the proposed swap heuristic) and by the *Site* heuristic (that uses a pure greedy method). These tests were performed in terrains 2, 3 and 4 using radius of interest 400

Table 3: Processing time (in seconds) of three methods: two parallel methods using GPU (*SparseSite* and *SiteGSM*) and a sequential one (*Site+*) to site observers on terrains with different sizes considering different radii of interest (R) to achieve some desired coverages (Ω).

Ter.	R	Ω	#Obs.	Processing Time (s)				
				SparseSite		*SiteGSM*		*Site+*
1	100	75%	36	1	(1017)	2	(509)	1017
		85%	44	1	(1599)	2	(800)	1599
		95%	56	2	(1767)	4	(883)	3533
	200	75%	9	0.5	(150)	0.2	(375)	75
		85%	12	0.5	(256)	0.4	(320)	128
		95%	15	0.7	(437)	0.8	(383)	306
	300	75%	4	0.4	(28)	0.1	(110)	11
		85%	5	0.4	(58)	0.2	(115)	23
		95%	7	0.5	(142)	0.4	(178)	71
2	200	75%	81	30	(5398)	76	(2131)	161951
		85%	97	42	(6725)	110	(2568)	282433
		95%	126	65	(7352)	173	(2762)	477855
	300	75%	36	19	(1737)	27	(1222)	33000
		85%	43	25	(2369)	39	(1518)	59221
		95%	54	37	(2887)	61	(1751)	106824
	400	75%	20	16	(708)	14	(809)	11321
		85%	25	18	(985)	20	(887)	17731
		95%	31	23	(1340)	27	(1141)	30813

Table 4: Processing time and number of observers used by the *SparseSite* heuristic (greedy with local search) and by the *Site* method (pure greedy heuristic). The last column presents the percentual difference in the number of observers used by the heuristics.

Ter.	*SparseSite*		*Site*		Difference
	#Obs.	Time(s)	#Obs.	Time(s)	
2	24	27	27	13	13%
3	102	381	112	95	10%
4	420	19011	461	2082	10%

cells and desired coverage 85%. Notice that *SparseSite* was able to achieve the same coverage as *Site* using, on average, 11% less observers. Of course, since the local search performs much more computation than a pure greedy method, it is much slower. However, the reduction in the number of observers may represent an important (economic) improvement, since observers may represent expensive facilities such as cell phone antennas.

Figure 5 compares the solution returned by the Greedy method with the solution obtained using the Greedy method with the *swap* heuristic. This siting was performed in Terrain 1, using radius of interest of 200 cells and the desired coverage was 75%. In this figure, we used a small terrain with high radius of interest and low desired coverage just to show the difference between the methods. Notice that the solution obtained using the *swap* heuristic used 9 observers, while the solution obtained using the purely Greedy method used 10 observers to reach the same minimum coverage.

6. CONCLUSIONS AND FUTURE WORK

This paper presented an efficient observer siting heuristic, that uses CUDA enabled GPUs to efficiently process high resolution terrains. The proposed method stores the viewsheds using a sparse-matrix format in order to reduce its memory usage and, as the tests showed, it was able to process terrains that could not be handled by previous parallel methods even in high-end GPUs. Also, tests in smaller terrains that can be processed by other methods showed that the use of processing techniques specially designed for sparse matrices led to a speedup of up to 2.7 times if compared against the fastest previous parallel implementation and up to 7352 if compared against the sequential method. This speedup was achieved not only by using GPU, but also due to the new strategies used to overlay the viewsheds using a fast sparse matrix multiplication algorithm and the dynamic programming.

These results are important in several applications like, for example, optimizing the position of communication towers

(a) Greedy method.

(b) Greedy method improved with the *swap* heuristic.

Figure 5: Comparing the solutions obtained using (a) the Greedy method with (b) the Greedy method improved with the *swap* local search.

or sensors such that a given percentual of a terrain is visually covered. By using the proposed parallel method, a terrain that would need more than 5 days to be processed using a sequential implementation could be processed in 65 seconds.

For future work, we intend to improve the performance and memory usage of the proposed implementation for situations where the viewshed data is not sparse (for example, when the observers have radii of interest closer to the terrain size) and, also, to perform experiments using the proposed local search in other heuristics such as GRASP and Simulating Annealing.

ADDITIONAL AUTHORS

Daniel Benedetti (Rensselaer Polytechnic Institute)
`daniel.n.benedetti@gmail.com`

ACKNOWLEDGEMENTS

This work was partially supported by FAPEMIG, CNPq, CAPES (Ciência sem Fronteiras) and NSF IIS-1117277.

7. REFERENCES

[1] M. V. A. Andrade, S. V. G. Magalhães, M. A. Magalhães, W. R. Franklin, and B. M. Cutler. Efficient viewshed computation on terrain in external memory. *GeoInformatica*, 15(2):381–397, 2011.

[2] B. Ben-Moshe. *Geometric Facility Location Optimization*. PHD thesis, Ben-Gurion University, Israel, Department of Computer Science, 2005.

[3] Y. Ben-Shimol, B. Ben-Moshe, Y. Ben-Yehezkel, A. Dvir, and M. Segal. Automated antenna positioning algorithms for wireless fixed-access networks. *Journal of Heuristics*, 13(3):243–263, 2007.

[4] A. Buluç, J. R. Gilbert, and C. Budak. Solving path problems on the GPU. *Parallel Comput.*, 36(5-6):241–253, June 2010.

[5] W. R. Franklin. Siting observers on terrain. In Springer-Verlag, editor, *In D. Richardson and P. van Oosterom editors, Advances in Spatial Data Handling: 10th International Symposium on Spatial Data Handling*, pages 109–120, 2002.

[6] W. R. Franklin and C. Ray. Higher isn't necessarily better: Visibility algorithms and experiments. In *Advances in GIS research: sixth international symposium on spatial data handling*, volume 2, pages 751–770. Edinburgh, 1994.

[7] Z. Li, Q. Zhu, and C. Gold. *Digital terrain modeling: principles and methodology*. CRC Press, 2005.

[8] G. O. López, F. Vázquez, I. García, and E. M. Garzón. Fastspmm: An efficient library for sparse matrix matrix product on GPUs. *Comput. J.*, 57(7):968–979, 2014.

[9] S. V. G. Magalhães, M. V. A. Andrade, and C. Ferreira. Heuristics to site observers in a terrain represented by a digital elevation matrix. In *GeoInfo*, pages 110–121, 2010.

[10] S. V. G. Magalhães, M. V. A. Andrade, and R. S. Ferreira. Using GPU to accelerate heuristics to site observers in DEM terrains. In *IADIS Applied Computing (AC 2011)*, pages 127–133. 2011.

[11] S. V. G. Magalhães, M. V. A. Andrade, and W. R. Franklin. An optimization heuristic for siting observers in huge terrains stored in external memory. In *Hybrid Intelligent Systems (HIS), 2010 10th International Conference on*, pages 135–140. IEEE, 2010.

[12] G. Nagy. Terrain visibility. *Computers & graphics*, 18(6):763–773, 1994.

[13] J. P. L. NASA. Shuttle Radar Topography Mission., 2013. Available at http://www2.jpl.nasa.gov/srtm (accessed on July 2014).

[14] NVIDIA. CUDA C programming guide. *NVIDIA Corporation, July*, 2014. Available at http://docs.nvidia.com/cuda (accessed on July 2014).

[15] G. C. Pena, M. V. A. Andrade, S. V. G. Magalhães, W. R. Franklin, and C. R. Ferreira. An improved parallel algorithm using GPU for siting observers on terrain. In *16th International Conference on Enterprise Information Systems (ICEIS-2014)*, pages 367–375, Lisbon, Portugal, 2014.

Table 5: Table of notations.

Symbol	Description
O	Observer
T	Target
O_b	Observer's base point
T_b	Target's base point
h	Height of an observer or target above terrain
R	Radius of interest of an observer
V	Viewshed of an observer
ω	Visibility index of an observer
$\mathcal{S}$	Set of observers
$\mathcal{V}$	Joint viewshed of a set of observers
Ω	Joint visibility index of a set of observers
$\mathcal{P}$	Set of candidate observers
n	Number of candidate observers
A	Set of candidate observers
S	Subset of A
k	Number of observers in S
S'	Neighbor solution of S
$i_1, \cdots, i_k$	Observers' Indices
$\oplus$	Union operation between two viewsheds
V_{i_k}	Viewshed of the observer k in S
$\mathcal{V}_{\overline{r}}$	Joint viewshed of all viewsheds in S except V_{i_r}
$vsize$	Number of points in each viewshed
$Vix[r][j]$	Joint visibility index of a solution replacing observer r with j
λ_r^-	Union between the viewsheds $V_{i_1} \cdots V_{i_{r-1}}$
λ_r^+	Union between the viewsheds $V_{i_{r+1}} \cdots V_{i_k}$