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

Guilherme C. Pena
Salles V. G. Magalhães
Marcus V. A. Andrade
W. Randolph Franklin
Chaulio R. Ferreira
Wenli Li

3rd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data
Introduction

- Problem: siting observers in a raster terrain in order to obtain an optimal visual coverage.
- Example: cover 95% of a terrain. How many and where to site observers to achieve this coverage?
Terrain visibility

- An observer is a point from which we wish to see or communicate with other points, called targets.
- Visibility depends on the radius of interest (R) of an observer and on the terrain topography.
Terrain visibility

- **viewshed** of an observer: set of terrain points whose corresponding targets are visible from it. Usually represented using a bit matrix.

- **visibility index/visible area** of an observer: number of visible targets.

- **joint viewshed** of a set of observers: union of the individual viewsheds.

Terrain visualization  Viewshed  Joint viewshed
Observer siting

- Observer siting: given a set of candidate observers, select the smallest subset of the candidates that is able to cover a minimum area.

- This problem is NP-Hard → usually solved using a heuristic.

- A greedy solution: Site method (Franklin 2002).

- Idea: greedily insert the observers in the solution until the target visibility index is reached.
Multiple observer siting

- The greedy solution is (mostly) not optimal.

- We propose a local search strategy (to try) to increase the terrain coverage preserving the number of observers selected → this may reduce the number of observers needed.

- It was used with the greedy heuristic, but it can be used as part of other heuristics to improve the solutions.

- Local search + greedy: improve each partial solution → less iterations.
Important concepts

- A neighbor solution of $S$ is a solution $S'$ where an observer in $S$ is replaced with another observer not in $S$.
- Local search: given a solution $S$, interactively improve $S$ by replacing it with its best neighbor solution.
- Stop when reach a solution without better neighbor.
For example: Suppose $P$ with 5 observers whose viewsheds are $V_1, V_2, ..., V_5$ and let $S=\{V_1, V_2, V_3\}$ be a partial solution. Thus, the neighbors of $S$ are:

- $S_1'$ = $\{V_2, V_3, V_4\}$
- $S_2'$ = $\{V_2, V_3, V_5\}$
- $S_3'$ = $\{V_1, V_3, V_4\}$
- $S_4'$ = $\{V_1, V_3, V_5\}$
- $S_5'$ = $\{V_1, V_2, V_5\}$
- $S_6'$ = $\{V_1, V_2, V_4\}$
Our propose – local search

- Challenge: for each neighbor solution, it is necessary:
  - to compute the overlap of all viewsheds;
  - to count the number of visible points;

- There are many neighbors: for 1000 (candidate) observers and a partial solution with 100 there are 90000 neighbors.

- This process is repeated in each iteration of the local search!
Local search: an efficient implementation

- The local search bottleneck is the computation of the visibility index of all neighbor solutions.

- Let \( P = \{p_1, \ldots, p_n\} \) be the candidate set and \( S = \{s_1, \ldots, s_k\} \) be a partial solution.

- The neighbors of \( S \) are

\[
S'_{ij} = S \setminus \{s_i\} \cup \{p_j\}
\]

for all \( i=1,\ldots,k \) and \( j=1,\ldots,n \) with \( i \neq j \) and \( p_j \notin S \)
Local search: an efficient implementation

- The visibility indices computation can be subdivided in two steps:

1. Create an array $B$ of size $k$ and for $i=1,\ldots,k$, store in $B[i]$ the joint viewshed of $S \setminus \{s_i\}$;

2. Create a matrix $Vix$ of size $k \times n$ and for each $i=1,\ldots,k$ and $j=1,\ldots,n$, with $j \neq i$, store in $Vix[i,j]$ the visibility index of the joint viewshed obtained overlapping $B[i]$ with the viewshed of the observer $p_j$. 

GPU parallel observer siting algorithm
Local search: an efficient implementation

- A straightforward implementation of step 1 is:

  for \( i \leftarrow 1 \) to \( k \) do
  for \( m \leftarrow 1 \) to \( k \) do
    if \( m \neq i \) then
      // overlap \( B[i] \) with \( S[m] \)
      \( B[i] \leftarrow B[i] \cup S[m] \)

- This code performs \( \Theta(k^2) \) overlapping operations;

- We can make it much better using dynamic programming.
Local search: an efficient implementation

- Suppose the partial solution $S$ has 5 observers, that is, $S = \{S_1, \ldots, S_5\}$.

- Then, the computation of $B$ would require the overlapping of the following viewsheds:

|---|----------|----------|----------|----------|----------|
Local search: an efficient implementation

- Suppose the partial solution $S$ has 5 observers, that is, $S = \{S_1, \ldots, S_5\}$.

- Then, the computation of $B$ would require the overlapping of the following viewsheds:

- The matrix with all $B$’s can be split in the following way:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$B[1]$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Local search: an efficient implementation

- The computation of the matrix storing all B’s can be rewritten as following:

\[
\begin{align*}
\end{align*}
\]

These two matrices can be computed separately using dynamic programming.

\[
L_1 = \emptyset \quad \text{and} \quad L_i = L_{i-1} \cup S_{i-1} \quad \text{for } i=2,\ldots,k
\]

\[
R_k = \emptyset \quad \text{and} \quad R_i = S_{i+1} \cup R_{i+1} \quad \text{for } i=k-1,\ldots,1
\]
Local search: an efficient implementation

Thus, the step 1 can be computed performing $\Theta(k)$ overlapping operations:

- $k$ to compute $L$;
- $k$ to compute $R$;
- $k$ to overlap $L$ and $R$
Local search: an efficient implementation

- In step 2, to compute the matrix $Vix$:
  - each joint viewshed stored in $B$ is overlapped with the viewshed of each candidate observer did not include in the solution yet;
  - the number of 1 bits in the resulting joint viewshed is counted.
Local search: an efficient implementation

- A straightforward implementation of step 2 is:

  for \( i \leftarrow 1 \) to \( k \) do
  for \( j \leftarrow 1 \) to \( n \) do
    \(/ / \) count the number of 1 bits in \( B[i] \cup P[j] \)
    for \( w \leftarrow 1 \) to \( V\text{size} \) do
      \( Vix[i,j] \leftarrow Vix[i,j] + (B[i,w] \lor P[j,w]) \)
Local search: an efficient implementation

- Which is equivalent to:

  
  for $i \leftarrow 1$ to $k$
  
  for $j \leftarrow 1$ to $n$
  
  // count the number of 1 bits in $B[i] \cup P[j]$
  
  for $w \leftarrow 1$ to $Vsize$
  
  $Vix[i,j] \leftarrow Vix[i,j] + (B[i,w] \text{ OR } P^T[w,j])$

- This code: similar to matrix multiplication.
- $X \rightarrow \text{ OR}$
- $\rightarrow$ Adapt a matrix multiplication algorithm.
Local search: an efficient implementation

- \( Vix[i,j] \leftarrow Vix[i,j] + (B[i,w] \text{ OR } P^T[w,j]) \)
- \( B \) is “multiplied” by \( P^T \)
- \( B[i] \): joint viewshed of \( S \setminus \{s_i\} \rightarrow \text{dense} \)
- \( P[j] \): viewshed of point \( j \rightarrow \text{sparse} \)
- For efficiency: sparse-dense MM!
Challenge

- 0 is the absorbing element in the multiplication operation.

\[
\begin{bmatrix}
1 & 1 & 1 \\
1 & 1 & 1
\end{bmatrix} \times \begin{bmatrix}
0 & 0 \\
0 & 0
\end{bmatrix} = \begin{bmatrix}
0 & 0 \\
0 & 0
\end{bmatrix}
\]

- ... but not in the OR operation.

- \[ \rightarrow \] sparse-dense MM algorithms cannot be directly used.

- Solution: compute the vix increment instead of the visibility index of the union.
Area increment

Before: \( Vix[i,j] = vix \) obtained when the i-th observer in the solution is replaced with the j-th candidate observer.
- \( Vix[i,j] \leftarrow Vix[i,j]+(B[i,w] \text{ OR } P^T[w,j]) \)

Now: \( Vix[i,j] = \) how much would the \( vix \) of \( B[i] \) increase if we add the j-th candidate observer.
- \( Vix[i,j] \leftarrow Vix[i,j]+((B[i,w] \text{ OR } P^T[w,j]) \text{ AND } \sim B[i,w]) \)

A "0" creates a null contribution.
Reducing the memory usage

- The B matrix stores the joint viewsheds → it is dense → may not fit in the GPU's memory.

- Proposed solution: divide the B matrix in smaller matrices $B_{a,b}$.

- In each step, compute $V_{ix_{a,b}}$ : area increment
Reducing the memory usage

- Challenge: compute $B_{a,b}$ efficiently.

|---|------|------|------|------|------|------|

GPU parallel observer siting algorithm
Reducing the memory usage

- Solution: compute the two rows before performing each dynamic programming step.
- Viewsheds are in GPU → fast.

We need this row

|------|------|------|------|------|------|

We need this row

GPU parallel observer siting algorithm
Results

- Our algorithm \textit{SparseSite} was implemented using CUDA and an efficient sparse-dense MM algorithm from the literature.
- Compared against: \textit{Site+} and \textit{SiteGSM}.
- Both are also based on the greedy strategy and use local search, but
  - \textit{Site+} uses a sequential (CPU) implementation. Does not use MM and dynamic programming.
  - \textit{SiteGSM}: does not represent the viewsheds using sparse matrices. Also, it does not divide the matrices.
Results

- The tests were executed on a computer with a GPU NVIDIA Tesla Kepler K20x (2688 cores) and CUDA 5.0.

- Terrains obtained from NASA STRM.

source: Nvidia.com
## Results

<table>
<thead>
<tr>
<th>Terrain</th>
<th>$R$</th>
<th>$\Omega$</th>
<th>#Obs.</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>75%</td>
<td>346</td>
<td></td>
<td>720</td>
</tr>
<tr>
<td></td>
<td>85%</td>
<td>410</td>
<td></td>
<td>1158</td>
</tr>
<tr>
<td>7500²</td>
<td>95%</td>
<td>517</td>
<td></td>
<td>1950</td>
</tr>
<tr>
<td></td>
<td>75%</td>
<td>87</td>
<td></td>
<td>279</td>
</tr>
<tr>
<td></td>
<td>85%</td>
<td>102</td>
<td></td>
<td>381</td>
</tr>
<tr>
<td></td>
<td>95%</td>
<td>126</td>
<td></td>
<td>610</td>
</tr>
<tr>
<td></td>
<td>75%</td>
<td>354</td>
<td></td>
<td>11830</td>
</tr>
<tr>
<td>15000²</td>
<td>85%</td>
<td>420</td>
<td></td>
<td>19011</td>
</tr>
<tr>
<td></td>
<td>95%</td>
<td>549</td>
<td></td>
<td>33863</td>
</tr>
</tbody>
</table>

- Large terrains.
- Site+: > 5 days
- SiteGSM: out of memory

GPU parallel observer siting algorithm
Results

<table>
<thead>
<tr>
<th>$n_r$</th>
<th>Time (sec)</th>
<th>Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4533</td>
<td>217</td>
</tr>
<tr>
<td>5</td>
<td>2069</td>
<td>304</td>
</tr>
<tr>
<td>10</td>
<td>2043</td>
<td>410</td>
</tr>
<tr>
<td>20</td>
<td>1939</td>
<td>621</td>
</tr>
<tr>
<td>40</td>
<td>1952</td>
<td>1043</td>
</tr>
<tr>
<td>80</td>
<td>1958</td>
<td>1887</td>
</tr>
<tr>
<td>160</td>
<td>1957</td>
<td>3577</td>
</tr>
<tr>
<td>260</td>
<td>1972</td>
<td>5688</td>
</tr>
</tbody>
</table>

- Memory usage vs time.
- Terrain: $7500^2$, Coverage: 95%
- Even keeping 5 rows in the memory → good performance.
- Smaller terrains.
- Table: time(s) and speedup.
- Up to 7000x of speedup over Site+.
- Up to 2.7 times faster than SiteGSM.
- Slower than SiteGSM using larger radius.

<table>
<thead>
<tr>
<th>Ter.</th>
<th>R</th>
<th>Ω</th>
<th>#Obs.</th>
<th>Processing Time (s)</th>
<th>SiteGSM</th>
<th>Site+</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SparseSite</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1201^2</td>
<td>100</td>
<td>75%</td>
<td>36</td>
<td>1 (1017)</td>
<td>2 (509)</td>
<td>1017</td>
</tr>
<tr>
<td></td>
<td></td>
<td>85%</td>
<td>44</td>
<td>1 (1599)</td>
<td>2 (800)</td>
<td>1599</td>
</tr>
<tr>
<td></td>
<td></td>
<td>95%</td>
<td>56</td>
<td>2 (1767)</td>
<td>4 (883)</td>
<td>3533</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>75%</td>
<td>9</td>
<td>0.5 (150)</td>
<td>0.2 (375)</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td></td>
<td>85%</td>
<td>12</td>
<td>0.5 (256)</td>
<td>0.4 (320)</td>
<td>128</td>
</tr>
<tr>
<td></td>
<td></td>
<td>95%</td>
<td>15</td>
<td>0.7 (437)</td>
<td>0.8 (383)</td>
<td>306</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>75%</td>
<td>4 (28)</td>
<td>1 (110)</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>85%</td>
<td>5</td>
<td>0.4 (58)</td>
<td>0.2 (115)</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td></td>
<td>95%</td>
<td>7</td>
<td>0.5 (142)</td>
<td>0.4 (178)</td>
<td>71</td>
</tr>
<tr>
<td>3601^2</td>
<td>75%</td>
<td>81</td>
<td>30 (5398)</td>
<td>76 (2131)</td>
<td>161951</td>
<td></td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>85%</td>
<td>97 (6725)</td>
<td>110 (2568)</td>
<td>282433</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>95%</td>
<td>126 (7352)</td>
<td>173 (2762)</td>
<td>477855</td>
<td></td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>85%</td>
<td>43</td>
<td>19 (1737)</td>
<td>27 (1222)</td>
<td>33000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>95%</td>
<td>54</td>
<td>37 (2887)</td>
<td>61 (1751)</td>
<td>106824</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>75%</td>
<td>20 (708)</td>
<td>16 (809)</td>
<td>11321</td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>85%</td>
<td>25</td>
<td>18 (985)</td>
<td>20 (887)</td>
<td>17731</td>
</tr>
<tr>
<td></td>
<td></td>
<td>95%</td>
<td>31</td>
<td>23 (1340)</td>
<td>27 (1141)</td>
<td>30813</td>
</tr>
</tbody>
</table>
Conclusion

- Fast implementation of a observer siting method.
- Based on a greedy strategy combined with a local search. Dynamic programming + GPU + sparse-dense matrix multiplication.
- Saves memory using sparse matrices and dividing the dense matrices.
- Can be used to improve other heuristics that solves other optimization problems.
Thank you!

Acknowledgements

source: Nvidia.com

source: wikipedia
Future work

- Develop parallel implementation using GPU to:
  - compute the viewshed of each observer;
  - replace the greedy strategy.