Universidade Federal de Viçosa (UFV)
Rensselaer Polytechnic Institute (RPI)

UFV

RPI

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

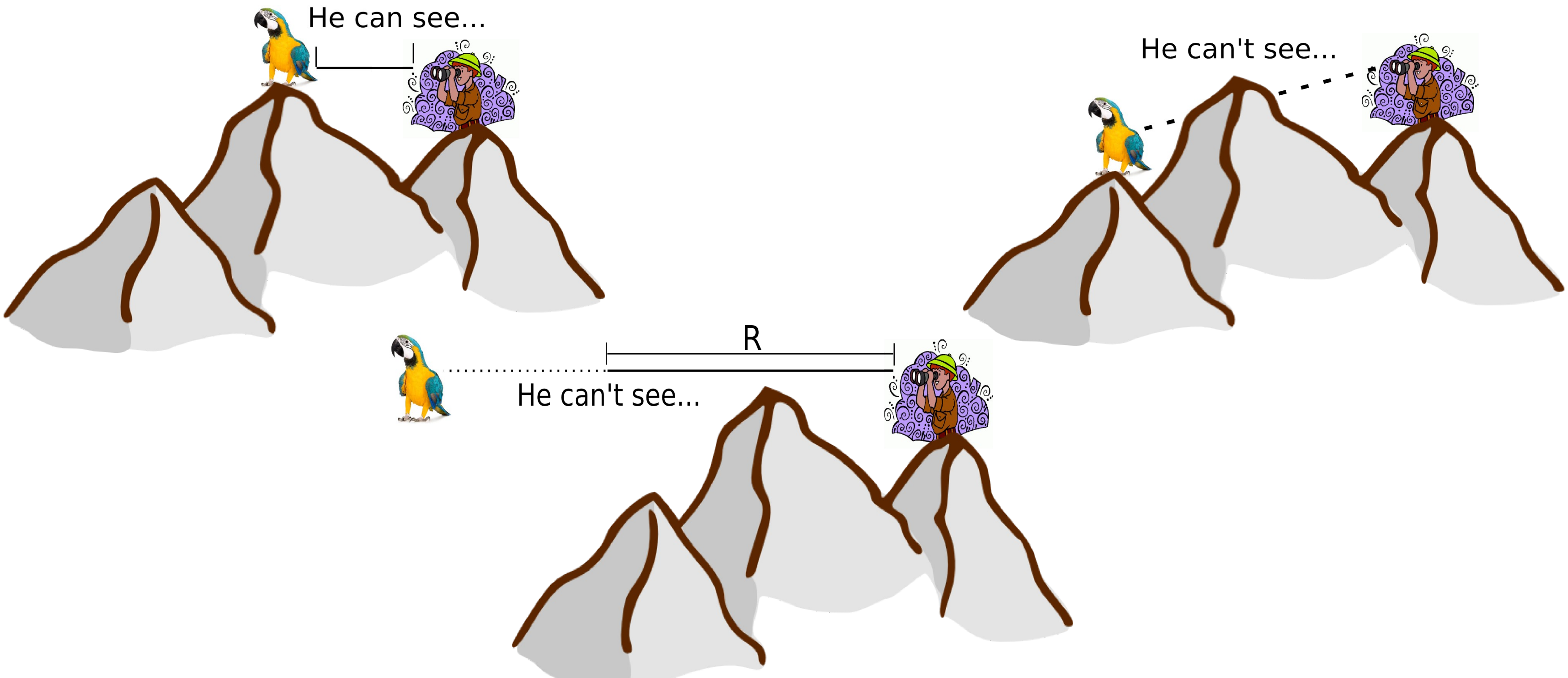
acm SIGSPATIAL

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?

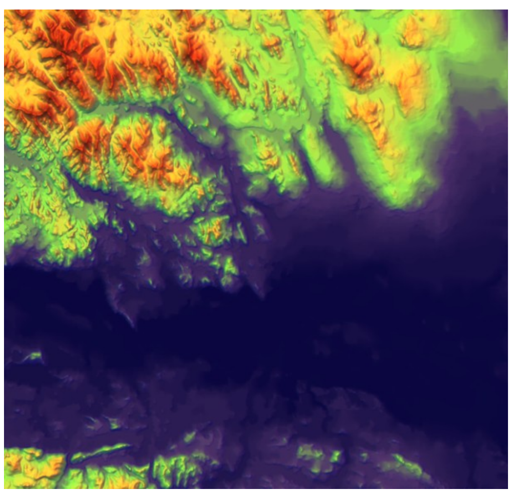
Terrain visibility

- 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.



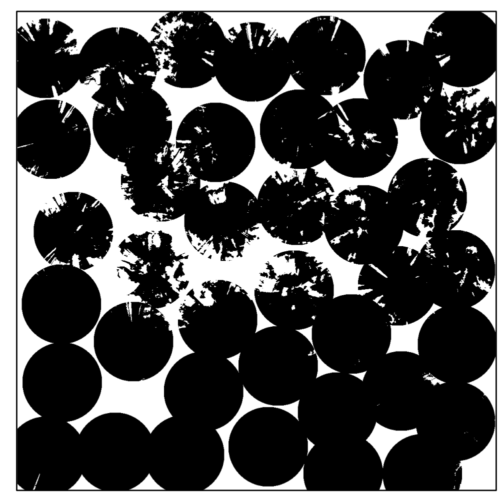
Terrain visibility

- *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

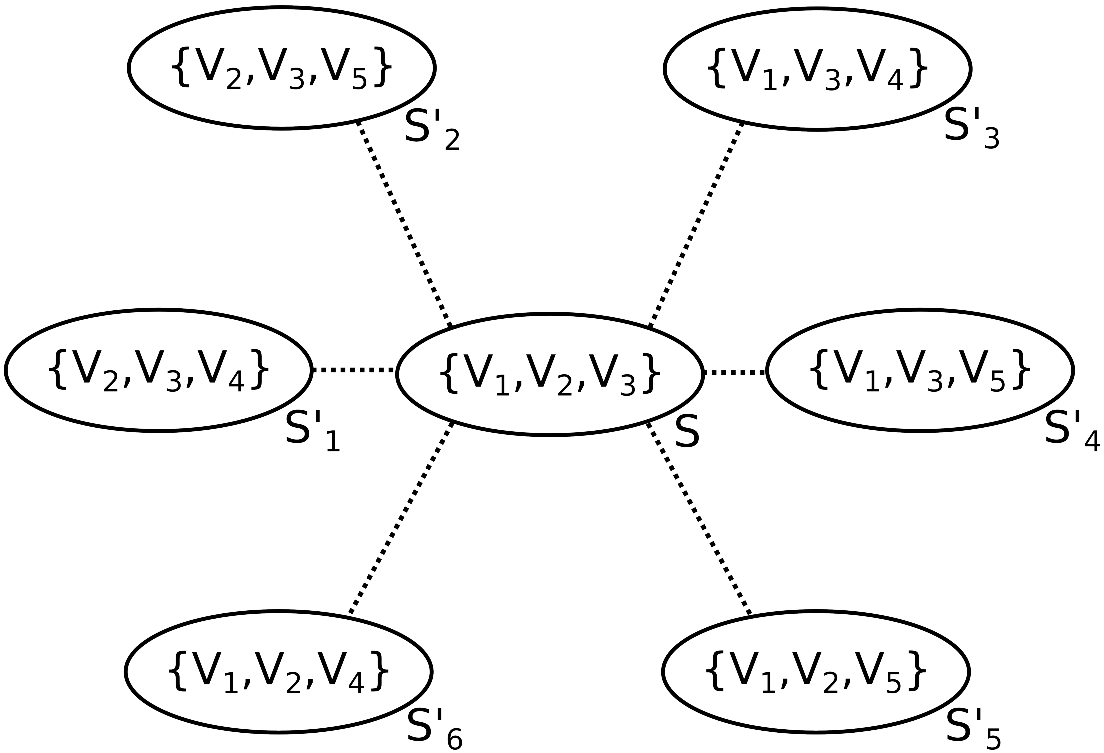
- The greedy solution is (mostly) not optimal.
- We propose a local search strategy (to try) to increase the terrain coverage preserving the number of observers selected → this may reduce the number of observers needed.
- It was used with the greedy heuristic, but it can be used as part of other heuristics 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.

Our propose – local search

- **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



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 $i=1,..,k$ and $j=1,..,n$ 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 $i=1,\dots,k$, store in $B[i]$ the joint viewshed of $S \setminus \{s_i\}$;

  ② Create a matrix Vix of size k x n and for each $i=1,\dots,k$ and $j=1,\dots,n$, 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.

Local search: an efficient implementation

- A straightforward implementation of step 1 is:

```
for i ← 1 to k do
  for m ← 1 to k do
    if m ≠ i then
      // overlap B[i] with S[m]
      B[i] ← B[i]  U 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:

B[1] =	S[2]	S[3]	S[4]	S[5]
B[2] =	S[1]	S[3]	S[4]	S[5]
B[3] =	S[1]	S[2]	S[4]	S[5]
B[4] =	S[1]	S[2]	S[3]	S[5]
B[5] =	S[1]	S[2]	S[3]	S[4]

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 … e overlapping of the following v… neds:

The matrix with all B's can be split in the following way

B[1] =	S[2]	S[3]	S[4]	S[5]
B[2] =	S[1]	S[3]	S[4]	S[5]
B[3] =	S[1]	S[2]	S[4]	S[5]
B[4] =	S[1]	S[2]	S[3]	S[5]
B[5] =	S[1]	S[2]	S[3]	S[4]

Local search: an efficient implementation

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

B[1]=	S[2]	S[3]	S[4]	S[5]
B[2]=	S[1]	S[3]	S[4]	S[5]
B[3]=	S[1]	S[2]	S[4]	S[5]
B[4]=	S[1]	S[2]	S[3]	S[5]
B[5]=	S[1]	S[2]	S[3]	S[4]

=

S[1]			
S[1]	S[2]		
S[1]	S[2]	S[3]	
S[1]	S[2]	S[3]	S[4]

L

\+

S[2]	S[3]	S[4]	S[5]
	S[3]	S[4]	S[5]
		S[4]	S[5]
			S[5]

R

- These two matrices can be computed separately using dynamic programming.

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

$$R_k = \Phi \ \text{ and } \ R_i = S_{i+1} \cup R_{i+1} \quad \text{for } i=k-1,\ldots,1$$

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 $Vsize$ do
            $Vix[i,j] \leftarrow Vix[i,j] + (B[i,w] \text{ OR } P[j,w])$

Vix of $S/\{s_i\} + p_j$

Points in $S/\{s_i\}$

Points in p_j

Local search: an efficient implementation

- Which is equivalent to:

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 $Vsize$ do
      $Vix[i,j] \leftarrow Vix[i,j]+(B[i,w] \text{ OR } P^T[w,j])$

- This code: similar to matrix multiplication.
- $X \rightarrow$ 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 $S \setminus \{s_i\}$ → dense
- P[j] : viewshed of point j → sparse
- → For efficiency: sparse-dense MM!

Challenge

- 0 is the absorbing element in the multiplication operation.

$$\begin{matrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{matrix} \times \begin{matrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{matrix} = \begin{matrix} 0 & 0 \\ 0 & 0 \end{matrix}$$

- … but not in the OR operation.
- →sparse-dense MM algorithms cannot be directly used.

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

Area increment

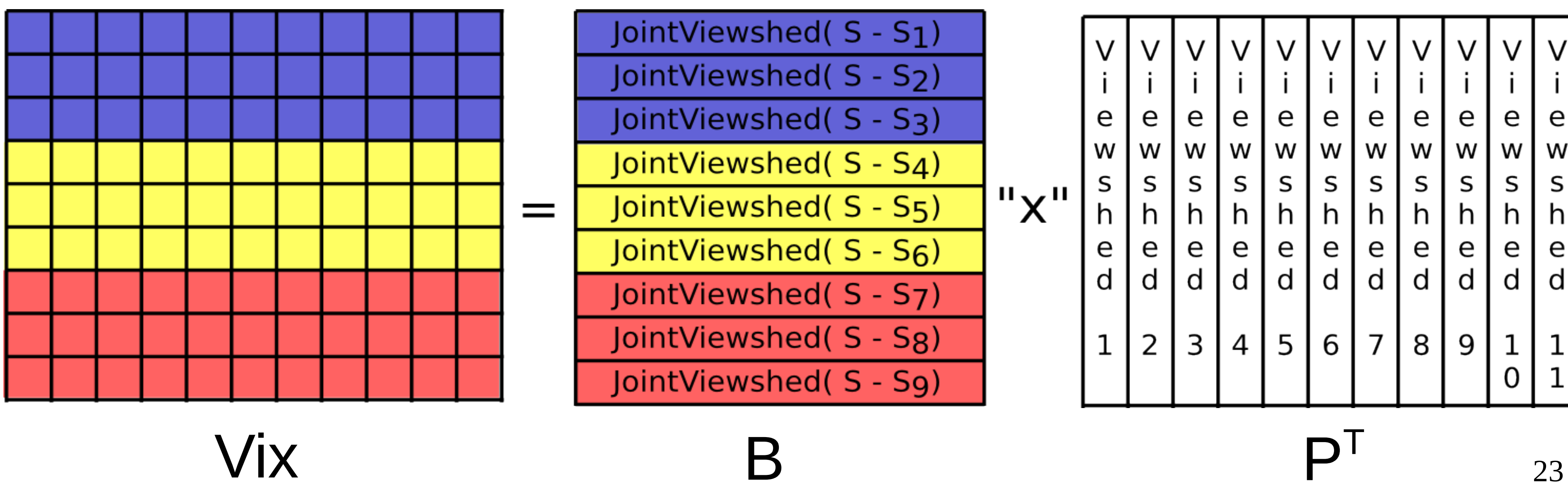
- Before: *Vix*[i,j] = *vix* obtained when the i-th observer in the solution is replaced with the j-th candidate observer.
  - $Vix[i,j] \leftarrow Vix[i,j]+(B[i,w] \text{ OR } P^T[w,j])$

- 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 $Vix_{a,b}$: area increment



Reducing the memory usage

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

B

B[1]	S[2]	S[3]	S[4]	S[5]	S[6]
B[2]	S[1]	S[3]	S[4]	S[5]	S[6]
B[3]	S[1]	S[2]	S[4]	S[5]	S[6]
B[4]	S[1]	S[2]	S[3]	S[5]	S[6]
B[5]	S[1]	S[2]	S[3]	S[4]	S[6]
B[6]	S[1]	S[2]	S[3]	S[4]	S[5]

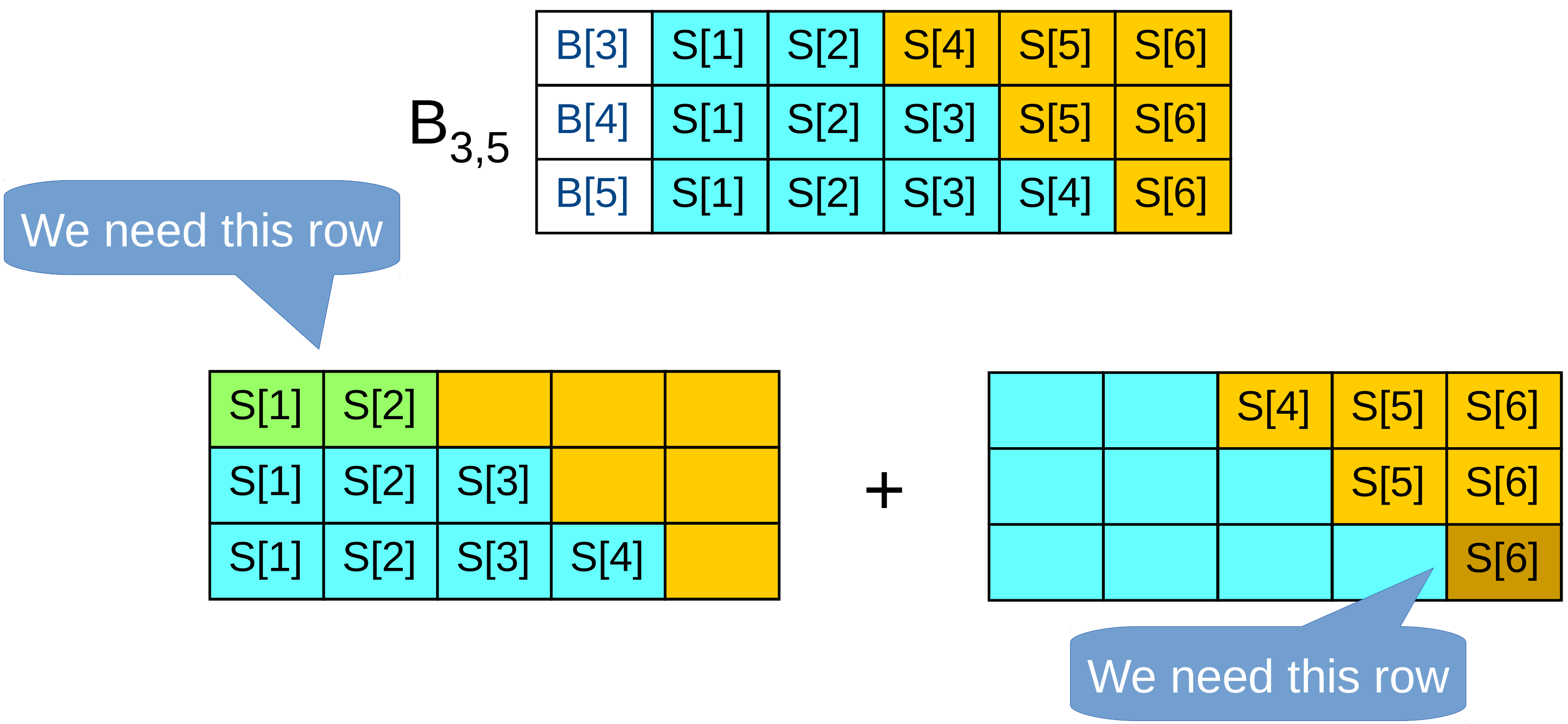
S[1]				
S[1]	S[2]			
S[1]	S[2]	S[3]		
S[1]	S[2]	S[3]	S[4]	
S[1]	S[2]	S[3]	S[4]	S[5]

+

S[2]	S[3]	S[4]	S[5]	S[6]
	S[3]	S[4]	S[5]	S[6]
		S[4]	S[5]	S[6]
			S[5]	S[6]
				S[6]

Reducing the memory usage

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



Results

- Our algorithm *SparseSite* was implemented using CUDA and an efficient sparse-dense MM algorithm from the literature.
- Compared against: *Site*+ and *SiteGSM.*
- Both are also based on the greedy strategy and use local search, but
  - *Site*+ uses a sequential (CPU) implementation. Does not use MM and dynamic programming.
  - *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

Terrain	R	Ω	#Obs.	Time(sec)
7500^2	200	75%	346	720
		85%	410	1158
		95%	517	1950
	400	75%	87	279
		85%	102	381
		95%	126	610
15000^2	400	75%	354	11830
		85%	420	19011
		95%	549	33863

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

Results

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

- Memory usage vs time.
- Terrain: 7500^2 , Coverage: 95%
- Even keeping 5 rows in the memory → good performance.

Ter.	R	Ω	#Obs.	Processing Time (s)				
				SparseSite		*SiteGSM*		*Site+*
1201^2	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
3601^2	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

- 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.

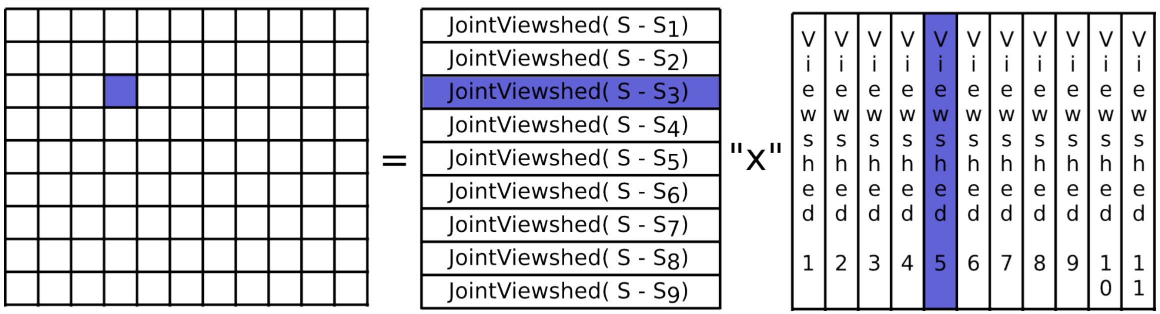
Conclusion

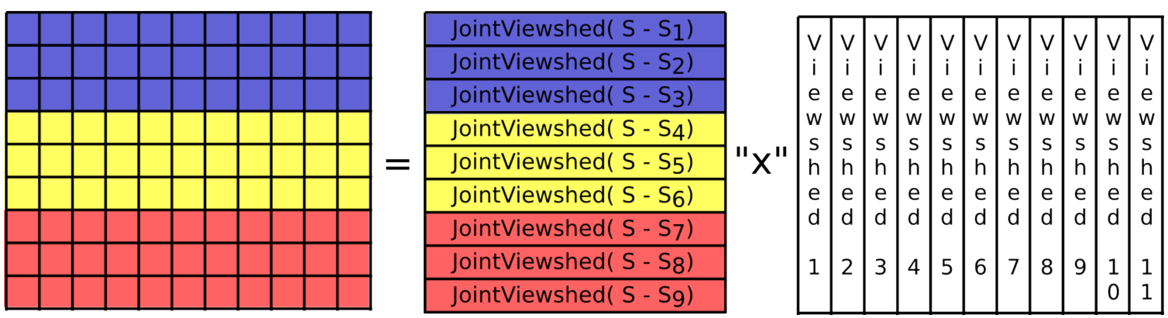
- Fast implementation of a observer siting method.
- Based on a greedy strategy combined with a local search. Dynamic programming + GPU + sparse-dense matrix multiplication.
- Saves memory using sparse matrices and dividing the dense matrices.
- Can be used to improve other heuristics that solves other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

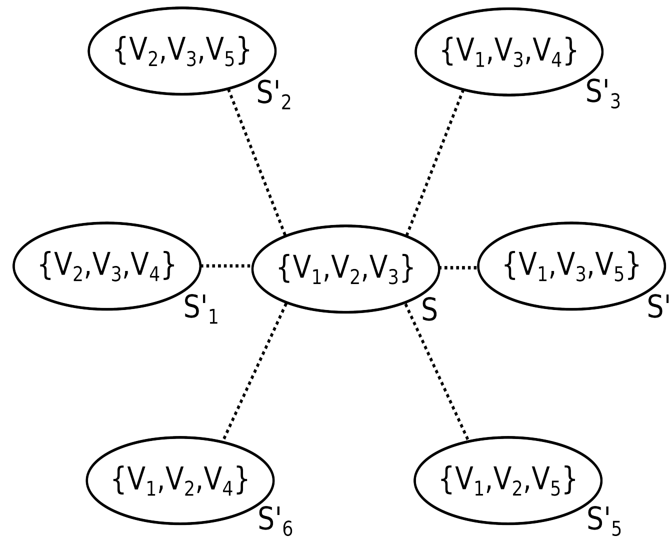
Thank you!

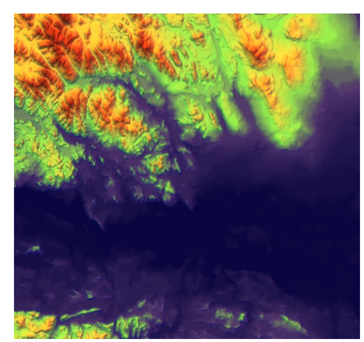
B[3]	S[1]	S[2]	S[4]	S[5]	S[6]
B[4]	S[1]	S[2]	S[3]	S[5]	S[6]
B[5]	S[1]	S[2]	S[3]	S[4]	S[6]

source: Nvidia.com









Acknowledgements

CAPES
CNPq Conselho Nacional de Desenvolvimento Científico e Tecnológico
FAPEMIG
NSF

Future work

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