GUILHERME DE CASTRO PENA

MÉTODO PARA O POSICIONAMENTO DE OBSERVADORES EM TERRENOS UTILIZANDO PROGRAMAÇÃO PARALELA EM GPU

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

VIÇOSA
MINAS GERAIS - BRASIL
2014

Ficha catalográfica preparada pela Biblioteca Central da Universidade Federal de Viçosa - Câmpus Viçosa

T
P397m
2014

Pena, Guilherme, 1988-
Método para o posicionamento de observadores em terrenos utilizando programação paralela em GPU / Guilherme Pena. – Viçosa, MG, 2014.
xi, 60f. : il. (algumas color.) ; 29 cm.

Orientador: Marcus Vinícius Alvim Andrade.
Dissertação (mestrado) - Universidade Federal de Viçosa.
Referências bibliográficas: f.57-60.

1. Programação paralela - Computação. 2. Visibilidade em Terrenos. 3. Sistemas de informação geográfica. 4. otimização combinatória. I. Universidade Federal de Viçosa. Departamento de Informática. Mestrado em Ciência da Computação. II. Título.

CDD 22. ed. 005.1

GUILHERME DE CASTRO PENA

MÉTODO PARA O POSICIONAMENTO DE OBSERVADORES EM TERRENOS UTILIZANDO PROGRAMAÇÃO PARALELA EM GPU

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Viçosa in partial fulfillment of the requirements for the degree of Master in Computer Science.

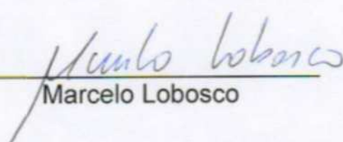
VIÇOSA
MINAS GERAIS - BRASIL
2014

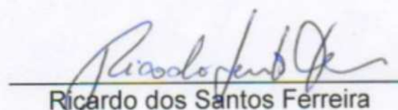
GUILHERME DE CASTRO PENA

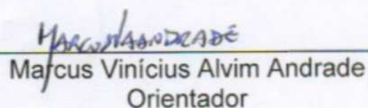
MÉTODO PARA O POSICIONAMENTO DE OBSERVADORES EM TERRENOS UTILIZANDO PROGRAMAÇÃO PARALELA EM GPU

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

APROVADA: 07 de agosto de 2014.


Marcelo Lobosco


Ricardo dos Santos Ferreira


Marcus Vinícius Alvim Andrade
Orientador

"O pessimista vê dificuldade em cada oportunidade, o otimista, a oportunidade em cada dificuldade."

(Winston Churchill)

Agradecimentos

Agradeço primeiramente a Deus pela benção e por todas as coisas boas que acontecem em minha vida. O mestrado foi uma delas, um aprendizado que guardarei para sempre. Foram períodos apertados, porém sempre tive o apoio de uma pessoa em especial, meu orientador Prof. Marcus Vinicius Alvim Andrade. Uma pessoa brilhante que tive a honra de trabalhar nos últimos quatro anos, desde minha iniciação científica. Obrigado professor pelo conhecimento, pela paciência, pelo profissionalismo, pela simpatia. Agradeço também ao Prof. Salles V. G. Magalhães por toda a sua ajuda, conhecimento e trabalho. Agradeço ao amigo Chaulio R. Ferreira e a todos amigos do grupo de pesquisa do prof. Marcus pela ajuda. Gostaria de agradecer também a todos colegas, funcionários e professores do departamento de informática em especial ao prof. José Luís Braga. Inclusive agradeço ao departamento por todo o apoio e excelentes condições de trabalho.

Agradeço aos órgãos de financiamento CAPES, CNPq, FAPEMIG, NSF e ao Prof. Franklin que tornaram possível o desenvolvimento dos meus projetos de pesquisa.

Em especial, agradeço a todos os meus familiares, meus pais José e Ângela por todo amor, apoio e incentivo e meu irmão André. A todos os meus amigos e colegas de curso que sempre me motivaram. Agradeço aos integrantes da banda "Só penna que vua" pela diversão sempre presente.

Agradeço a todos vocês que construíram comigo mais este sonho. Do fundo do meu coração, muito obrigado!

Sumário

Lista de Figuras vi

Lista de Tabelas viii

Lista de Abreviaturas e Siglas ix

Resumo x

Abstract xi

1 Introdução geral 1

1.1 O Problema do Posicionamento de Observadores 3
1.2 Objetivos 4
1.3 Resultados obtidos 4

2 Algoritmo paralelo usando GPU para o posicionamento de observadores em terrenos 7

2.1 Introdução 7
2.2 Referencial Teórico 8
2.2.1 Visibilidade 8
2.2.2 CUDA e programação paralela de propósito geral 9
2.3 A heurística proposta 9
2.3.1 A busca local *Swap* 10
2.3.2 Implementação eficiente da heurística *Swap* 11
2.4 Resultados experimentais 12
2.5 Conclusão 13

3 An Improved Parallel Algorithm using GPU for Siting Observers on Terrain 15

3.1 Introduction 15

3.2 Background . . . 17
3.2.1 Terrain Visibility Definitions . . . 17
3.2.2 Parallel Programming Using General Purpose GPU . . . 18
3.3 Related Work . . . 19
3.4 The site method . . . 20
3.5 Observer siting in GPU . . . 21
3.5.1 The Local Search - Swap . . . 22
3.5.2 An Efficient Swap Heuristic Implementation . . . 23
3.6 Experimental Results . . . 28
3.7 Conclusion . . . 29

4 An efficient GPU multiple-observer siting method based on sparse-matrix multiplication 33
4.1 Introduction . . . 33
4.2 Terrain Visibility Definitions . . . 35
4.3 Related Work . . . 36
4.4 The proposed method . . . 38
4.4.1 The Local Search - Swap . . . 38
4.4.2 An Efficient Swap Heuristic Implementation . . . 41
4.4.3 Using a sparse-dense matrix multiplication algorithm . . . 43
4.4.4 Reducing the swap heuristic's memory usage . . . 45
4.5 Experimental Results . . . 46
4.6 Conclusions and future work . . . 51

5 Conclusões gerais e trabalhos futuros 55

Referências Bibliográficas 57

Lista de Figuras

1.1 Tipos de amostragem de pontos e formas de representação de MDEs. Fonte: [Câmara et al., 2001] . . . 2
  (a) Amostragem Irregular . . . 2
  (b) Amostragem Regular . . . 2
  (c) Malha Triangular . . . 2
  (d) Grade Regular . . . 2

3.1 Visibility queries using a line of sight. Source: [Magalhães et al., 2011b] . 17
3.2 Comparison between GPUs and CPUs structures. Source: [NVIDIA, 2014] 19
3.3 Given $A = \{V_1, V_2, V_3, V_4, V_5\}$, S' are the neighbors of the solution $S = \{V_1, V_2, V_3\}$. . . . 23
3.4 Matrix illustrating $\mathcal{V}_{\overline{r}}$ in a solution with $k = 5$ observers. . . . 25
3.5 Matrix Multiplication Algorithm. . . . 27
  (a) Matrices on global memory. . . . 27
  (b) Iterative load in the GPU's shared memory. . . . 27
  (c) Writing back to the global memory. . . . 27
3.6 The blocks are labeled as following: if all the cells in the block are not visible, the block is labeled with 0; otherwise with 1. . . . 28
  (a) Original V matrix. . . . 28
  (b) Mask-matrix over V. . . . 28
3.7 Observers sited in terrain 1201x1201 with desired coverage 25% and radii of interest: 100 (a), 200 (b) and 300 (c). The observers are indicated by yellow triangles and visible points are showed in red. . . . 30

4.1 Visibility queries using a line of sight. Source: [Magalhães et al., 2011b] . 35
4.2 Given $V = \{V_1, V_2, V_3, V_4, V_5\}$, each S'_i is obtained performing a swap in $S = \{V_1, V_2, V_3\}$. . . . 39
4.3 Matrix illustrating $\mathcal{V}_{\overline{r}}$ in a solution with $k = 5$ observers. . . . 41

4.4 ELLPACK-R matrix format representation: (a) an example showing how the matrix is packed; (b) how the viewshed matrix (from Figure 3.6a) is packed using this representation. 44
(a) Source: [Wafai et al., 2009] 44
(b) 44
4.5 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^+. 46
4.6 Processing-time and memory usage of *SparseSite* as a function of the number of rows (n_r) of the dense matrix that is kept in the memory. . . 49
4.7 Processing time (in seconds) of the three methods (*SparseSite*, *SiteGSM* and *Site+*) to site observers with radius of interest (R) of 200 (a), 300 (b) and 400 (c) cells on the terrain with 3601^2 cells. For clarity purposes, the y-scale is logarithmic. 53
(a) $R = 200$. 53
(b) $R = 300$. 53
(c) $R = 400$. 53
4.8 Comparison of a solution obtained by the greedy method against a solution obtained for the greedy method improved with the *swap* local search. 54
(a) Greedy method. 54
(b) Greedy method improved with the *swap* heuristic. 54

Lista de Tabelas

1.1 Comparação dos algoritmos descritos com relação às estratégias adotadas e detalhes de implementação. 6

2.1 Tempos de execução da heurística convencional (em CPU) e da heurística proposta; neste caso, são apresentados os tempos para obtenção do *viewshed* acumulado (Υ), da área visível e tempo total incluindo operações de entrada e saída. 13

3.1 Processing time (in seconds) of three methods: two parallel methods using GPU (*SiteGSM* and *SiteGPU*) 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). 31

3.2 Table of notations. 32

4.1 Processing time of *SparseSite* in large terrains. 48

4.2 Processing-time and memory usage of *SparseSite* as a function of the number of rows (n_r) of the dense matrix that is kept in the memory. These tests were performed on Terrain 3, using radius of interest equal to 200 cells and desired coverage of 95%. 48

4.3 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). The speedup of each method (compared against *Site+*) is shown in parenthesis. 50

4.4 Comparison of the processing time and number of observers used by the *SparseSite* heuristic (greedy with local search) and by the *Site* method (greedy heuristic). The last column presents the percentual difference in the number of observers used by the heuristics. 51

Lista de Abreviaturas e Siglas

CPU	-	*Central Processing Unit*
CUDA	-	*Compute Unified Device Architecture*
GPGPU	-	*General Purpose Graphic Processing Units*
GPU	-	*Graphics Processing Unit*
MDE	-	Modelo Digital de Elevação
MDT	-	Modelo Digital de Terreno
RSG	-	*Regular Square Grid*
SIG	-	Sistemas de Informação Geográfica
SIMD	-	*Single Instruction Multiple Data*
SM	-	*Stream Multiprocessors*
SRTM	-	*Shuttle Radar Topography Mission*
TIN	-	*Triangulated Irregular Network*

Resumo

PENA, Guilherme de Castro, M.Sc., Universidade Federal de Viçosa, agosto de 2014. **Método para o posicionamento de observadores em terrenos utilizando programação paralela em GPU.** Orientador: Marcus Vinícius Alvim Andrade.

Este trabalho apresenta um algoritmo eficiente, chamado *SparseSite*, para solucionar o problema do posicionamento de observadores em terrenos. Este problema é uma aplicação do conceito de visibilidade que consiste em determinar quais pontos do terreno são visíveis a partir de um ponto específico, denominado *observador*, e o conjunto de pontos visíveis por este observador é chamado *mapa de visibilidade* ou *viewshed*. Assim, o objetivo do problema é selecionar um conjunto de observadores cujos os *viewsheds* otimizem a cobertura do terreno, com aplicações nas áreas de telecomunicações, planejamento ambiental, monitoramento militar, entre outras. Segundo Nagy [1994], este problema é NP-Difícil e portanto, não se conhece um algoritmo eficiente que encontre a sua solução ótima. Assim, em geral, este problema é solucionado usando métodos heurísticos. Porém, mesmo as soluções aproximadas podem demandar um longo tempo de processamento devido ao grande volume de dados a ser analisado. O algoritmo descrito neste trabalho adota estratégias de programação paralela voltadas para arquiteturas de placas gráficas (GPUs) e além disso, permite o processamento de grandes volumes reorganizando os dados de entrada do problema e capacitando o usuário a gerenciar o uso de memória pela GPU. Ele é uma extensão do método *Site* proposto por Franklin [2002]. A heurística utilizada encontra soluções melhores do que as encontradas pelo método *Site*, isto é, usando um número menor de observadores. Os resultados experimentais mostraram que, comparado aos principais algoritmos descritos na literatura, o nosso método se mostrou muito mais eficiente do que eles, sendo mais de 7000 vezes mais rápido do que o método que não utiliza nenhuma técnica de melhoria e mais de 20 vezes mais rápido do que o método paralelo anterior. Além disso, foram realizados testes do *SparseSite* para processar volumes de dados que não puderam ser processados pelos outros métodos devido à limitações de memória ou por demandarem muito tempo de processamento. Por fim, comparado ao nosso algoritmo anterior *SiteGSM*, o *SparseSite* é quase 3 vezes mais rápido e usa 10% da memória usada por ele.

Abstract

PENA, Guilherme de Castro, M.Sc., Universidade Federal de Viçosa, August, 2014.
Method for siting observers on terrains using GPU parallel programming.
Adviser: Marcus Vinícius Alvim Andrade.

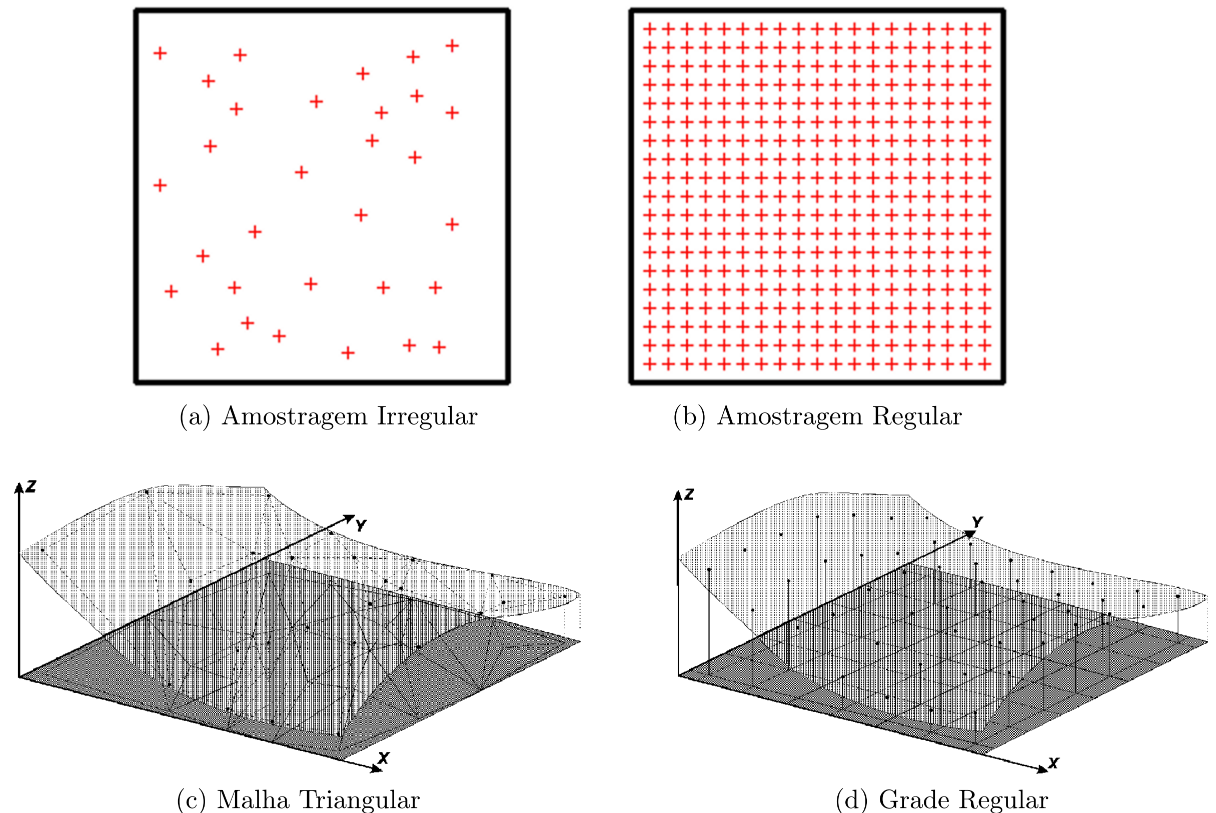
This work presents an efficient algorithm , named *SparseSite*, to siting observers on terrains. This problem concerns visibility, that is, determining the set of points that are visible from a particular point, called observer, and the set of visible points from this observer is called *viewshed*. Thus, the goal of this problem is to select a set of observers in order to "optimally" cover the terrain. The applications include telecommunications, environmental planning, military monitoring and so forth. According to Nagy [1994], the siting problem is NP-Hard and, therefore, there is no known efficient algorithm to find its optimal solution. Thus, in general, it is used a heuristic 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. The algorithm described in this work adopts parallel programming strategy for graphics processing units (GPUs) and furthermore, it allows the processing of large data volumes rearranging the input data of the problem and enabling the user to manage the memory usage by the GPU. This algorithm extends the method *Site* proposed by Franklin [2002] and the heuristic used finds even better solutions than those found by the method *Site*. The experimental results showed that, compared against the mainly algorithms described in previous works, our method proved to be more efficient than all of them, more than 7000 times faster than the sequential method without improvement techniques and 20 times faster than a previous parallel method. Additionally, some tests of the *SparseSite* were performed to process volumes of data which cannot be processed by the other methods due to memory limitations or to demand a lot of processing time. Lastly, compared against our previous algorithm *SiteGSM*, the *SparseSite* is almost 3 times faster and uses 10% of the amount of memory used by it.

1. Introdução geral

A modelagem digital de terrenos desempenha um importante papel nos Sistemas de Informação Geográfica (SIG) em diversas questões tais como visibilidade, hidrografia, roteamento, entre outras [Goodchild et al., 2005]. Em particular, o conceito de visibilidade em terrenos consiste em determinar quais pontos do terreno são visíveis a partir de um ponto específico, denominado *observador*, e o conjunto de pontos visíveis por este observador é chamado *mapa de visibilidade* ou *viewshed*. O observador pode estar localizado a uma certa altura acima do solo. Telecomunicações, planejamento ambiental, monitoramento militar são exemplos de importantes aplicações práticas que envolvem este conceito [Franklin and Ray, 1994; Li et al., 2005; Nagy, 1994; Andrade et al., 2011]. Dentre estas aplicações, pode-se destacar o problema do posicionamento de observadores, onde o objetivo é selecionar um conjunto de observadores que otimize a cobertura sobre o terreno. Estes observadores podem representar torres de radio, TV, telefonia móvel, Internet [Hrovat et al., 2010], ou câmeras ou torres de monitoramento [Ben-Moshe, 2005; Ben-Shimol et al., 2007]. Por exemplo, se um observador representar uma torre de telefonia celular, seu *viewshed* representaria as áreas do terreno onde se espera que um usuário do serviço de telefonia consiga obter sinal diretamente a partir dessa torre e assim, o problema do posicionamento poderia ser tratado de duas formas, determinar o número mínimo de torres de celular necessárias para cobrir uma certa região ou determinar a maior área possível coberta por um dado número de torres.

Em geral, os modelos digitais de terreno (MDTs) usados por estas aplicações possuem informações sobre a elevação da superfície terrestre. Segundo Câmara et al. [2001], a aquisição de dados geográficos para a construção desses modelos pode ser feita através da amostragem de pontos espaçados de forma irregular ou regular, Figuras 1.1a e 1.1b. Dessa maneira, a superfície da Terra é representada de forma aproximada utilizando um modelo digital de elevação (MDE), que armazena as elevações dos pontos amostrados. As amostras são armazenadas em estruturas de dados definidas de forma a possibilitar uma manipulação eficiente pelos algoritmos de análise contidos nos Sistemas de Informação Geográfica. No caso de amostras irregularmente espaçadas, normalmente utilizam-se estruturas de dados de malha triangular, denominadas *triangulated irregular networks* (TINs), Figura 1.1c. Para o caso de amostras regularmente espaçadas é possível utilizar estruturas de dados

mais simples, como grades regulares (*regular square grids* - RSGs) que consistem em matrizes que armazenam as elevações dos pontos amostrados, Figura 1.1d.



(a) Amostragem Irregular (b) Amostragem Regular

(c) Malha Triangular (d) Grade Regular

Figura 1.1: Tipos de amostragem de pontos e formas de representação de MDEs. Fonte: [Câmara et al., 2001]

Não há um consenso sobre qual destas representações é a mais adequada e existem vários trabalhos discutindo esta questão [Kumler, 1994; Floriani et al., 1999]. O que se pode dizer é que uma grade regular é mais fácil de analisar e é mais precisa considerando dados amostrados a altas resoluções. No entanto, ela requer mais espaço de memória. Por outro lado, uma TIN tem uma precisão restrita e requer algoritmos mais complexos. É importante ressaltar que a escolha de uma forma de representação específica não resulta em uma restrição relevante na prática, uma vez que existem métodos eficientes para a conversão entre as diversas formas de representação [Li et al., 2005]. Neste trabalho será considerada a representação de grades regulares devido à sua simplicidade e principalmente porque há uma enorme quantidade de dados disponíveis neste formato. Além disso, os algoritmos propostos neste trabalho foram baseados em outros algoritmos [Franklin, 2002; Magalhães et al., 2010; Magalhães et al., 2011a] que também trabalham com esta representação.

1.1 O Problema do Posicionamento de Observadores

Como mencionado anteriormente, o problema do posicionamento de observadores consiste em selecionar um conjunto de observadores que otimize a "cobertura" sobre um terreno. Conforme demonstrado em Nagy [1994], este problema é NP-Difícil e portanto, não se conhece um algoritmo eficiente que encontre a sua solução ótima. Dessa forma, geralmente são usados métodos heurísticos para se obter uma solução aproximada. Contudo, mesmo a obtenção de soluções aproximadas para este problema de otimização pode demandar um longo tempo de processamento uma vez que é necessário processar um grande volume de dados de alta resolução. Por exemplo, a base de dados SRTM (*Shuttle Radar Topography Mission*) [Rabus et al., 2003], criada no ano de 2000 e que ainda é amplamente usada por pesquisadores, possui em torno de 10 *terabytes* de dados terrestres disponíveis com resoluções de 30 e 90 metros. Atualmente, novas tecnologias de sensoriamento remoto são capazes de amostrar elevações com resoluções de 1 metro ou até mesmo submétricas gerando matrizes de elevação ainda maiores. Por exemplo, uma região de $500km \times 500km$ amostrada à 1 metro gera um terreno de 500 *gigabytes* aproximadamente.

Sendo assim, as aplicações em SIGs demandam o desenvolvimento de novas técnicas para processar esse volume de dados. Uma técnica bem sucedida, que é explorada por este trabalho, é o projeto de algoritmos paralelos baseados em unidades de processamento gráfico para propósito geral (*General Purpose Graphic Processing Units* - GPGPUs). Novas placas gráficas (GPUs) fornecem recursos que permitem o desenvolvimento de algoritmos bastante eficientes em um amplo domínio de aplicações [NVIDIA, 2014].

Dentre os algoritmos encontrados na literatura que solucionam o problema do posicionamento de observadores, destacam-se o método *Site* [Franklin, 2002] e os métodos *Site+* [Magalhães et al., 2010], *SiteGPU* [Magalhães et al., 2011a] e *EMSite* [Magalhães et al., 2011b] que são extensões do algoritmo *Site* adotando diferentes estratégias, porém no processamento de grandes volumes de dados, alguns deles possuem restrições práticas com relação ao uso de memória e outros demandam longos tempos de processamento. Mais detalhes desses métodos serão descritos posteriormente.

Para contornar esses problemas, nos Capítulos 2, 3 e 4 são apresentados os algoritmos que caracterizam a evolução do nosso trabalho. Conforme demonstrado pelos resultados experimentais, a técnica final obtida baseada em programação paralela

se mostrou eficiente quando comparada aos métodos disponíveis na literatura para processar maiores volumes de dados. Além disso, a possibilidade de gerenciar o uso da memória de placas gráficas permite que o método aqui proposto seja executado em placas de custo relativamente mais baixo, que possuem menores quantidades de memória disponível.

1.2 Objetivos

O objetivo geral deste trabalho foi o desenvolvimento de um método eficiente capaz de solucionar o problema de posicionamento de observadores. O método deve considerar que grandes volumes de dados deverão ser manipulados, de modo que deverá gerenciar adequadamente os recursos disponíveis no computador, em particular a memória. Para alcançar este objetivo geral, destacam-se alguns objetivos específicos:

- Propor e implementar um algoritmo que utilize as modernas arquiteturas paralelas de GPUs para realizar o processamento de forma mais eficiente;
- Realizar a análise de complexidade dos algoritmos desenvolvidos;
- Avaliar experimentalmente os algoritmos desenvolvidos, comparando-os com outros métodos encontrados na literatura.

1.3 Resultados obtidos

Os Capítulos de 2 a 4 são apresentados na forma de artigos e descrevem os resultados obtidos neste trabalho que são baseados no método *Site*, proposto por Franklin [2002], e nos métodos *Site+* [Magalhães et al., 2010] e *SiteGPU* [Magalhães et al., 2011a] que também correspondem a extensões do método *Site*. A comparação dos algoritmos é apresentada na Tabela 1.1.

Resumidamente, o método *Site* obtém uma solução aproximada baseada na estratégia gulosa para solucionar o posicionamento de observadores onde o objetivo é obter o número mínimo de observadores para alcançar uma determinada cobertura do terreno. Este método inicializa a solução como vazia e a cada iteração seleciona o observador (ainda não incluído na solução) que gera o maior aumento na cobertura total da sub-solução corrente.

A partir deste método, Magalhães et al. propuseram os métodos *Site+* [Magalhães et al., 2010] e *SiteGPU* [Magalhães et al., 2011a] que são extensões do método

Site onde a ideia básica é incluir uma busca local cujo objetivo é verificar se a troca de um observador já selecionado com algum outro ainda não selecionado leva a um aumento da cobertura. Ao final desta verificação, a busca local seleciona a melhor troca, isto é, aquela que gera o maior aumento. Como demonstrado por Magalhães et al. [2010], apesar de gastar mais tempo de processamento pela inclusão da busca local, esta nova estratégia permite que se use um número menor de observadores para atingir a mesma taxa de cobertura. Esta redução pode representar uma economia significativa visto que um observador pode ser uma instalação de alto custo (por exemplo, uma torre de celular).

Este trabalho descreve uma nova implementação paralela usando GPU da estratégia baseada na busca local [Magalhães et al., 2010]. O Capítulo 2 corresponde ao artigo "*Algoritmo paralelo usando GPU para o posicionamento de observadores em terrenos*" apresentado no GeoInfo 2013 (*XIV Brazilian Symposium on GeoInformatics*) [Pena et al., 2013]. Esse artigo descreve o algoritmo *SiteGSM* (*Site using GPU shared memory*) que destaca-se pela definição de uma nova implementação usando programação dinâmica executada na CPU em uma etapa intermediária do processo e usando a hierarquia de memórias da GPU na etapa posterior, com uso da memória compartilhada e da memória global da placa, atingindo maior eficiência do que os outros métodos. Nesse artigo, o problema do posicionamento foi tratado de modo a obter a maior cobertura possível atingida por um conjunto com um dado número de observadores. O *SiteGSM* foi comparado experimentalmente com o método *Site+* [Magalhães et al., 2010] que propõe uma implementação convencional dessa mesma estratégia usando apenas a CPU. Os resultados mostraram que o *SiteGSM* chegou a ser, em muitos casos, mais de 200 vezes mais rápido do que o método *Site+*.

O capítulo 3 corresponde ao artigo "*An Improved Parallel Algorithm using GPU for Siting Observers on Terrain*" apresentado no ICEIS 2014 (*16th International Conference on Enterprise Information Systems*) [Pena et al., 2014a]. Este artigo descreve uma segunda versão do método *SiteGSM* com algumas modificações importantes. Na primeira delas, a etapa de programação dinâmica (PD) que anteriormente era executada em CPU, passou a ser processada em GPU. A outra modificação se fez na etapa posterior à PD para reduzir o número de operações que estavam sendo realizadas na GPU sobre dados da entrada que não influenciavam no resultado. Estas modificações aumentaram a eficiência do método. Nesse artigo, o problema do posicionamento foi tratado de forma a minimizar o número de observadores necessários para atingir uma dada cobertura sobre o terreno. O novo *SiteGSM* foi comparado experimentalmente com o método *Site+* e o método *SiteGPU*, que

implementa algumas rotinas usadas pela heurística em GPU e mantém todos os dados na memória global da placa. Os resultados mostraram que o novo algoritmo *SiteGSM* chegou a ser, em alguns casos, mais de 20 vezes mais rápido do que o método *SiteGPU* e mais de 1200 vezes mais rápido do que o método *Site+*.

Finalmente, o capítulo 4 se refere ao artigo "*An efficient GPU multiple-observer siting method based on sparse-matrix multiplication*" submetido à ACM SIGSPATIAL 2014 (*22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*) [Pena et al., 2014b]. Neste artigo é apresentado o algoritmo *SparseSite* que se baseia no uso de matrizes esparsas para representar um conjunto de *viewsheds* que é considerado esparso e na adaptação de um algoritmo para multiplicação destas matrizes. Essa adaptação realiza sobreposição de matrizes e depois uma contagem para determinar a sua área total de visibilidade. Além disso, para reduzir a quantidade de memória usada pelo método, foi adotada a estratégia de particionar o processamento da matriz que armazena a sobreposição dos *viewsheds* da sub-solução. Essa ideia proposta no *SparseSite* fez com que o algoritmo mantivesse a eficiência do método *SiteGSM* usando uma quantidade muito menor de memória e assim, permitiu-se o processamento de maiores volumes de dados. Ele foi comparado experimentalmente com o método *Site+* e com o método anterior *SiteGSM*. Os resultados mostraram que o novo algoritmo *SparseSite* chegou a ser, em alguns casos, quase 3 vezes mais rápido do que o método *SiteGSM* e mais de 7000 vezes mais rápido do que o método *Site+*. Além disso, é importante mencionar que o novo método *SparseSite* atinge estes resultados usando em média 10% da memória usada pelo método *SiteGSM*.

Tabela 1.1: Comparação dos algoritmos descritos com relação às estratégias adotadas e detalhes de implementação.

Algoritmo	Estratégia		Implementação			
	Gulosa	Gulosa com Busca Local	Somente CPU	CPU e GPU	Programação Dinâmica	Programação Dinâmica em GPU
Site	x		x			
Site+		x	x			
SiteGPU		x		x		
SiteGSM		x		x	x	
SparseSite		x		x		x

2. Algoritmo paralelo usando GPU para o posicionamento de observadores em terrenos[1]

Abstract

This paper presents an efficient method for sitting a set of observers to maximize the coverage of a given terrain. It is based on an efficient implementation of a local search heuristic using dynamic programming and GPU parallel programming. The tests showed that the proposed method can be more than 200 times faster than the conventional implementation (with no use of dynamic programming and GPU parallel programming).

Resumo

Este artigo apresenta um método eficiente para o posicionamento de um conjunto de observadores de modo a maximizar a cobertura de um dado terreno. A estratégia proposta se baseia na implementação eficiente de uma heurística de busca local utilizando programação dinâmica e programação paralela (em GPU). Os testes mostraram que este método chega a ser mais de 200 vezes mais rápido que a implementação convencional (sem o uso de programação dinâmica e paralela).

2.1 Introdução

Um importante grupo de aplicações na área de Sistemas de Informação Geográficas envolve o conceito de visibilidade em terrenos, que consiste em determinar os pontos do terreno que são visíveis a partir de um dado ponto (*observador*). Este problema possui muitas aplicações práticas, tais como telecomunicações, planejamento ambiental, e monitoramento militar [Ben-Shimol et al., 2007; Bespamyatnikh et al., 2001]. Um problema importante relacionado à visibilidade é o posicionamento de

[1] Neste capítulo é apresentada uma primeira versão do método *SiteGSM*. Nele está incluído o artigo "*Algoritmo paralelo usando GPU para o posicionamento de observadores em terrenos*", apresentado no GeoInfo 2013 (*XIV Brazilian Symposium on GeoInformatics*) [Pena et al., 2013].

observadores para maximizar a "cobertura do terreno", sendo que estes "observadores" podem ser câmeras, torres de vigilância ou de telefonia móvel, etc. [Ben-Shimol et al., 2007; Franklin and Vogt, 2006; Eidenbenz, 2002]. Conforme demonstrado em Nagy [1994] este problema é NP-Difícil e portanto, não se conhece um algoritmo eficiente para resolvê-lo de maneira exata.

Mas, até mesmo a obtenção de soluções aproximadas para este problema de otimização demanda um alto tempo de processamento, principalmente ao processar grandes volumes de dados. Uma forma de reduzir esse tempo de processamento consiste em desenvolver algoritmos paralelos baseados nas unidades de processamento gráfico para propósito geral (*GPGPUs*), que estão presentes na maioria das placas gráficas atuais.

Este trabalho apresenta uma implementação paralela utilizando *GPU* de uma heurística para solucionar o problema do posicionamento de observadores em um terreno representado por um *MDE raster*. Mais precisamente, a heurística proposta visa maximizar a área coberta no terreno por um dado número de observadores.

2.2 Referencial Teórico

2.2.1 Visibilidade

Um *terreno* é uma representação da elevação da superfície terrestre em uma determinada região que, neste trabalho, será representado por uma matriz de elevação (ou *MDE*) que armazena a elevação de amostras do terreno regularmente espaçadas [Felgueiras, 2001].

Um *observador* é um ponto do espaço que "deseja" visualizar ou se comunicar com outros pontos do espaço chamados de *alvos*. Tanto o observador como o alvo podem estar a uma certa altura acima do terreno. Existem vários algoritmos para o cálculo da visibilidade e, em geral, eles adotam que um alvo T é *visível* a partir de um observador O se, e somente se, T estiver dentro do raio de interesse de O e não houver nenhum ponto do terreno bloqueando o segmento de reta que conecta O a T.

O *problema de posicionamento de observadores* consiste em posicionar um conjunto de observadores de modo a maximizar o número de pontos visíveis no *viewshed* acumulado destes observadores [Young-Hoon Kim et al., 2004; Magalhães et al., 2010]. Este problema é NP-Difícil [Nagy, 1994] e possui várias aplicações práticas como, por exemplo, otimizar o posicionamento de torres de observação, sistemas de radares ou torres de telefonia celular.

2.2.2 CUDA e programação paralela de propósito geral

Atualmente, uma técnica de programação paralela que tem sido muito utilizada é a *GPGPU* [NVIDIA, 2014], que consiste em utilizar os vários núcleos de processamento presentes em *GPUs* para processar dados. O modelo de programação *CUDA* facilita o desenvolvimento de aplicativos capazes de aproveitar o poder de processamento das *GPUs* da NVIDIA, que são compostas por vários processadores do tipo *SIMD* e possibilitam a execução de milhares de *threads* de forma paralela.

A arquitetura *CUDA* permite a execução de código tanto na *CPU* quanto na *GPU*. Assim, o processador principal (*CPU*) pode executar trechos de código que envolvam menos paralelismo e maior quantidade de estruturas de controle de fluxo de execução (que normalmente não são processadas de forma eficiente em arquiteturas *SIMD*) e designar para a execução na *GPU* funções que possam ser aplicadas de forma paralela sobre diferentes elementos de dados. Dessa forma, a *GPU* é utilizada como um co-processador que é capaz de executar certos tipos de tarefas de forma mais eficiente do que a *CPU*.

2.3 A heurística proposta

A solução proposta neste artigo consiste em implementar uma heurística de busca local (*Swap*) de forma eficiente utilizando *CUDA* e programação dinâmica. Mais precisamente, a heurística (*Swap*) será utilizada para aumentar a área visível de soluções previamente obtidas utilizando o método *Site* [Franklin and Vogt, 2006], que é baseado em uma heurística gulosa. Assim, dada uma solução obtida pelo método *Site* (ou por qualquer outro método), o método proposto tenta melhorar esta solução realizando operações de busca local baseadas em trocas (*Swap*) de observadores (*viewsheds*) para aumentar o número de pontos visíveis no *viewshed* acumulado, mantendo fixo o número de observadores utilizados na solução.

É importante ressaltar que em Magalhães et al. [2011a] é apresentada uma solução, usando programação paralela em GPU, para uma variação do problema de posicionamento de observadores onde o objetivo é "minimizar" o número de observadores selecionados para alcançar uma taxa de cobertura do terreno. Embora os problemas sejam semelhantes, as soluções (e implementações) adotadas são bem diferentes.

2.3.1 A busca local *Swap*

Dado um conjunto de n observadores candidatos, seja $V = \{V_1, \cdots V_n\}$ o conjunto com os respectivos *viewsheds* e seja S um subconjunto de V contendo k *viewsheds* representando uma solução para o problema de posicionamento de observadores. Assim, o objetivo da heurística *Swap* é realizar uma busca local para maximizar o número de pontos visíveis no *viewshed* acumulado de k observadores selecionados dentre os n candidatos.

Mais especificamente, a ideia da heurística *Swap* consiste em verificar iterativamente todas as soluções vizinhas à solução S e, a cada passo, substituir S pela solução vizinha com maior área visível. Dada uma solução S, as vizinhas de S são todas as soluções S' onde exatamente um dos *viewsheds* em S' é diferente de um *viewshed* de S. Por exemplo, se $V = \{V_1, V_2, V_3, V_4\}$ e $S = \{V_1, V_2, V_3\}$, as soluções vizinhas a S são: $\{V_1, V_2, V_4\}$, $\{V_1, V_4, V_3\}$, $\{V_4, V_2, V_3\}$. O processo de trocar a solução atual pela melhor vizinha é repetido até ser obtida uma solução que não possua melhores vizinhas.

Assim, considerando cada *viewshed* representado por uma matriz linearizada de inteiros (onde os números 0 e 1 representam, respectivamente, pontos não visíveis e visíveis), a heurística *Swap* pode ser implementada (sequencialmente) da seguinte forma: dados os *viewsheds* candidatos $V_1, \cdots V_n$, seja S uma solução composta por k *viewsheds*, isto é, $S = \{V_{i_1}, \cdots, V_{i_k}\}$ cujo *viewshed* acumulado é dado por $V_{i_1} \oplus \cdots \oplus V_{i_k}$ onde $\oplus$ representa a união de dois *viewsheds*[2]. Além disso, seja Υ_r o *viewshed* acumulado envolvendo todos os *viewsheds* em S exceto o *viewshed* V_{i_r}. Daí, em cada iteração da heurística são geradas todas as soluções vizinhas a S e a melhor delas é selecionada para ser a solução corrente utilizada na próxima iteração. As soluções vizinhas a S são geradas fazendo-se $\Upsilon_r \oplus V_j$ para $r = 1 \cdots k$ e $j = 1 \cdots n$.

A etapa que demanda maior tempo de processamento na heurística *Swap* é o cálculo da área visível para cada solução vizinha. O Algoritmo 1 apresenta o pseudocódigo dessa etapa, onde $Area[r][j]$ representa o número de pontos visíveis em $\Upsilon_r \oplus V_j$ e, portanto, essa matriz pode ser utilizada para se obter qual é a melhor vizinha de S.

Por motivos de eficiência, neste trabalho os *viewsheds* foram codificados em palavras de 64 *bits* (onde cada palavra representa a visibilidade de 64 pontos no terreno). Com isso, a união de *viewsheds* e o cálculo da área visível podem ser realizadas utilizando, respectivamente, o operador **or** de *bits* e funções de conta-

[2]A união de dois *viewsheds* pode ser realizada aplicando-se a operação **or** binária entre cada ponto de um *viewshed* e o ponto correspondente do outro.

Algoritmo 1 Calcula a matriz $Area$; *VSize* é o número de pontos em cada *viewshed*.

1: $Area[1..k][1..n] \leftarrow \{\{0\}\}$
2: **for** $r = 1 \rightarrow k$
3: **for** $j = 1 \rightarrow n$
4: **for** $w = 1 \rightarrow VSize$
5: $Area[r][j] \leftarrow Area[r][j] + (\Upsilon[r][w]$ **or** $V[j][w])$

gem de população, que são operações disponibilizadas no *hardware* da maioria dos computadores atuais.

2.3.2 Implementação eficiente da heurística *Swap*

Note que para gerar os valores de Υ_r com $r = 1 \cdots k$, o algoritmo descrito na seção 2.3.1 realiza $k \times n = \Theta(kn)$ operações $\oplus$ de união de *viewsheds* (sendo que cada união envolve Θ*(VSize)* posições na matriz dos *viewsheds*).

Conforme descrito a seguir, a eficiência dessa etapa do algoritmo pode ser melhorada consideravelmente utilizando-se programação dinâmica. Para simplificar a notação, sejam $\nu_1, \cdots \nu_k$ os k *viewsheds* selecionados para formar a solução S, ou seja, $\nu_1, \cdots \nu_k = V_{i_1} \cdots V_{i_k}$. Assim, $\Upsilon_r = (\nu_1 \oplus \nu_2 \oplus ... \oplus \nu_{r-1}) \oplus (\nu_{r+1} \oplus \nu_{r+2} \oplus \cdots \oplus \nu_k)$. Sejam $\lambda_r = (\nu_1 \oplus \nu_2 \oplus ... \oplus \nu_{r-1})$ e $\overline{\lambda}_r = (\nu_{r+1} \oplus \nu_{r+2} \oplus \cdots \oplus \nu_k)$. Note que λ_r pode ser obtido com base na seguinte equação de recorrência: $\lambda_1 = \emptyset$ e $\lambda_r = (\nu_1 \oplus \nu_2 \oplus ... \oplus \nu_{r-1}) = (\lambda_{r-1} \oplus \nu_{r-1})$. De forma análoga, temos que $\overline{\lambda}_k = \emptyset$ e $\overline{\lambda}_r = (\nu_{r+1} \oplus \nu_{i+2} \oplus \cdots \oplus \nu_k) = (\nu_{r+1} \oplus \overline{\lambda}_{r+1})$. Daí, $\Upsilon_r = \lambda_r \oplus \overline{\lambda}_r$. Essas relações de recorrência foram utilizadas para implementar um algoritmo baseado em programação dinâmica para o cálculo da matriz que representa os valores de Υ, sendo que este algoritmo realiza $\Theta(k)$ uniões de *viewsheds*.

Após o cálculo de Υ, o próximo passo consiste em utilizar esses valores para realizar uma iteração da heurística *Swap*. Uma forma de se implementar essa etapa na GPU consiste em manter todos os *viewsheds* na memória global da GPU e, então, utilizar cada *thread* para calcular cada um dos elementos da matriz *Area*. Essa implementação convencional não aproveitaria de forma eficiente todo o poder da GPU, pois os acessos seriam realizados na memória global, que é mais lenta do que outras memórias, como a memória compartilhada. Como a capacidade da memória compartilhada é muito pequena, ela não é capaz de armazenar todos os dados e, portanto, é necessário adotar uma estratégia de dividir o processamento para que os cálculos possam ser realizados por partes.

Note que, no Algoritmo 1, as matrizes *Area*, Υ e V são acessadas de forma

similar ao padrão de acesso utilizado em um algoritmo de multiplicação de matrizes, onde as matrizes Υ e V^T são multiplicadas para gerar a matriz $Area$. A única diferença entre o Algoritmo 1 e um algoritmo de multiplicação de matrizes é que, no algoritmo de multiplicação, a operação $Area[r][j] \leftarrow Area[r][j] + (\Upsilon[r][w]\ \mathbf{or}\ V[j][w])$ (linha 5) é substituída pela operação $Area[r][j] \leftarrow Area[r][j] + \Upsilon[r][w] \times V^T[w][j]$.

Assim, podemos utilizar as técnicas de otimização adotadas na implementação em GPU de algoritmos de multiplicação de matrizes. Neste trabalho, utilizamos (adaptamos) o algoritmo de multiplicação de matrizes descrito em NVIDIA [2014], substituindo a operação de multiplicação por uma operação de **or** binário seguida por uma operação de contagem de população. Esse algoritmo divide as matrizes em blocos, que são carregados iterativamente na memória compartilhada à medida em que o processo de multiplicação é realizado. Com isso, a maior parte dos acessos realizados pelo algoritmo é feita na memória compartilhada, que é muito mais rápida do que a memória global.

2.4 Resultados experimentais

Para avaliar o desempenho do método proposto, foram implementadas duas versões da heurística *Swap*: uma versão utilizando o método convencional para cálculo de Υ, sem uso de programação dinâmica, e o algoritmo sequencial para o cálculo da matriz $Area$ e uma outra versão utilizando programação dinâmica em CPU para o cálculo de Υ e programação paralela (em GPU) para a obtenção da matriz $Area$. Essas duas versões foram implementadas em C++ e compiladas, respectivamente, com os compiladores *g++* 4.6.4 e *nvcc* 4.0 (ambos com nível de otimização -O3). Foram realizados testes em um computador com processador Intel Xeon E5-2687 3.1GHz, 128GiB de memória RAM e GPU NVidia Tesla Kepler K20x, que possui 2688 núcleos de processamento CUDA, 6GiB de memória global e CUDA 5.0.

As implementações foram testadas em dois terrenos obtidos a partir do projeto *SRTM* da NASA, com dimensões 1201×1201 e 3601×3601 células. Os testes para cada terreno foram executados variando-se o número total de pontos candidatos e o número de observadores selecionados. A Tabela 2.1 apresenta os resultados obtidos.

Os resultados apresentados na Tabela 2.1 representam todas as iterações da heurística até atingir um máximo local, ou seja, até ser obtida uma solução que não possui melhores vizinhas. Além disso, é apresentado o ganho (*Speedup*) da implementação proposta em relação à implementação convencional das heurísticas e como pode-se notar, os ganhos são melhores à medida que as configurações de número de

Tabela 2.1: Tempos de execução da heurística convencional (em CPU) e da heurística proposta; neste caso, são apresentados os tempos para obtenção do *viewshed* acumulado (Υ), da área visível e tempo total incluindo operações de entrada e saída.

Ter.	#Can.	#Obs.	Cob.	Tempo de processamento (em seg.)						
				$\Upsilon(s)$		Area (s)		Total (s)		
				Conv.	P.D.	CPU	GPU	Conv.	Proposta	
1201 × 1201	500	16	25%	0.1	0.1	17.4	0.1	17.6	0.9	(20x)
		32	39%	1.3	0.1	98.7	0.5	100	1.6	(63x)
		64	47%	9.2	0.3	351	1.5	360	3.4	(106x)
	1000	32	55%	1.1	0.1	175	0.7	177	1.9	(93x)
		64	83%	10.7	0.4	829	3.1	839	5.2	(161x)
		128	95%	94	1.6	3363	12	3457	18	(192x)
		256	98%	640	5.5	11129	39.1	11769	56.8	(207x)
3601 × 3601	500	16	1%	0.4	0.1	52.3	0.4	53	2.6	(20x)
		32	1%	2.6	0.1	183	0.9	186	3.6	(52x)
		64	2%	18	0.6	635	2.8	654	6.9	(95x)
	1000	32	2%	2.2	0.2	314	1.3	317	5.1	(62x)
		64	3%	23.9	0.8	1689	6.2	1713	12.8	(134x)
		128	5%	175	3	6083	21.7	6259	35.2	(178x)
		256	7%	1375	12	24114	83.7	25489	126	(202x)
	2000	32	2%	2.2	0.2	636	2.3	639	8.7	(73x)
		64	4%	13.4	0.5	1880	6.7	1895	14.2	(133x)
		128	6%	192	3.3	13381	46.9	13575	64	(212x)
		256	10%	1320	11.5	46008	159	47329	203	(234x)

observadores (#Obs), número de *viewsheds* candidatos (#Can) e quantidade de células do terreno aumentam. A coluna Cob apresenta a cobertura atingida pelo conjunto final de observadores.

2.5 Conclusão

Foi proposta uma heurística de busca local *Swap* eficiente para a solução do problema de posicionamento de observadores com o objetivo de maximizar a cobertura no terreno. O método proposto combina técnicas de programação dinâmica com programação paralela em CUDA e chega a ser, em muitos casos, mais de 200 vezes mais rápido do que uma implementação convencional.

A eficiência obtida é importante não só para o processamento de grandes quantidades de dados, mas também para a implementação de outras heurísticas (como GRASP e ILS) [Magalhães et al., 2010], que normalmente executam várias vezes

heurísticas de busca local (como a *Swap*).

Como trabalhos futuros, pretende-se analisar o impacto da heurística proposta em outros métodos que realizam buscas locais. Além disso, pretende-se também desenvolver outras métodos de busca local para o problema de posicionamento de observadores.

3. An Improved Parallel Algorithm using GPU for Siting Observers on Terrain[1]

Abstract

This paper presents an efficient method to determine a set of observers (that is, where to site them) such that a given percentage of a terrain is visually covered. Our method extends the method proposed in Franklin [2002] including a local search heuristic efficiently implemented using dynamic programming and GPU parallel programming. This local search strategy allows to achieve a higher coverage using the same number of observers as the original method and thus it is possible to obtain a given coverage using a smaller number of observers. It can be an important improvement since an observer can represent an expensive facility such as a telecommunication tower. The proposed method performance was compared with that of other methods and the tests showed that it can be more than 1200 times faster than the sequential implementation (with no use of dynamic programming and no GPU parallel programming) and, also, more than 20 times faster than a previous parallel method presented in Magalhães et al. [2011a].

3.1 Introduction

A large amount of high-resolution geographic data has become available because of the recent advances in remote sensing. Thus, the development of advanced techniques to process these data has been required by Geographic Information Science (GIS) [Laurini and Thompson, 1992]. The Earth surface elevation (terrain) data are usually represented approximately by a digital elevation matrix (DEM) that stores the elevations of regularly sampled terrain points. Elevations of intermediate points are usually approximated using some interpolation process [Li et al., 2005].

An important group of GIS applications concerns visibility, i.e., determining the set of points that are visible from some particular point, called observer. The observer can be located at some height above the terrain. These applications include

[1] Neste capítulo é apresentado o artigo "*An Improved Parallel Algorithm using GPU for Siting Observers on Terrain*", apresentado no ICEIS 2014 (*16th International Conference on Enterprise Information Systems*) [Pena et al., 2014a].

telecommunications, environmental planning, autonomous vehicle navigation and military monitoring [Franklin and Ray, 1994; Li et al., 2005; Nagy, 1994; Andrade et al., 2011]. One important related problem is the siting of a given number 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 [Ben-Moshe, 2005; Ben-Shimol et al., 2007]. As described in Nagy [1994], this is an NP-Hard problem and, therefore, there is no known efficient algorithm to find its optimal solution.

However, even obtaining approximate solutions for this optimization problem demands a long processing time, particularly when processing large quantities of data. One way to reduce this processing time is to design parallel algorithms based on general purpose graphics processing units (GPGPUs), which are present in most current graphics cards.

This paper deals with an instance of the multiple observers 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. In Franklin [2002], the author presented a solution for this problem based on greedy strategy and, in this paper, we extend that method including a local search heuristic based on a swapping strategy to achieve a better terrain coverage using the same number of observers. As the main contribution of this paper, this heuristic was implemented in parallel using Graphics Processing Units (GPUs) and dynamic programming.

The extended method proposed in this paper, named *SiteGSM* (from Site using GPU's shared memory), was compared with some other methods for siting observers on terrain such as Franklin [2002]; Franklin and Vogt [2006]; Magalhães et al. [2010]; Magalhães et al. [2011a] and, as the tests showed, our method achieves a better performance than all of them: it is faster than Magalhães et al. [2010]; Magalhães et al. [2011a] (in some cases, more than 20 times faster than both) or it can use a smaller number of observers than Franklin [2002]; Franklin and Vogt [2006] to cover the terrain (which may represent an important improvement since the observer can be an expensive facility as, for example, a communication tower).

3.2 Background

3.2.1 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 point of the earth surface 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*. As usual, 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 that appear in this work are shown in table 3.2.

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 3.1. In this Figure, T_1 is visible from O but T_2 is not.

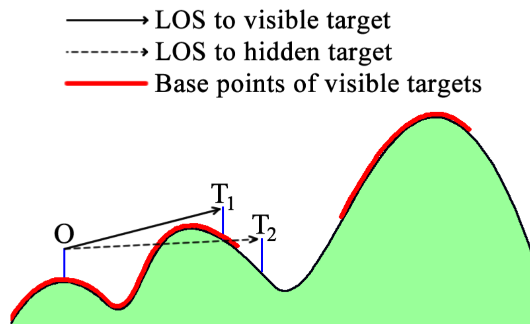


Figure 3.1: Visibility queries using a line of sight. Source: [Magalhães et al., 2011b]

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 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 area of the terrain that is seen by at least one observer [Franklin and Ray, 1994].

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 [Nagy, 1994] 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 terrain.

3.2.2 Parallel Programming Using General Purpose GPU

The programming architectures that allow using GPU units' parallel computing power (as, for example, the Compute Unified Device Architecture (CUDA) [NVIDIA, 2014]) have led to the development of many algorithms that achieve high computation performances.

CUDA has made possible the development of algorithms to solve time-consuming problems using the large number of parallel multiprocessors as well as the high memory bandwidth provided by GPUs. To accomplish high-performance computing, it is necessary to develop parallel algorithms that are totally or partially executed on the GPU.

The CUDA-enabled graphics cards are composed of multiple processors, more specifically, Single Instruction Multiple Data (SIMD) processors called Stream Multiprocessors (SMs), which allow the execution of multiple parallel threads. Thus, GPU processors can efficiently execute instructions involving many operations with data parallelism, i.e., when the same operation is applied to different data.

According to NVIDIA, GPUs can provide greater processing power than CPUs because they are specialized in performing parallel tasks involving many calculations. On the other hand, the CPUs are designed to perform tasks involving execution flow

control and data cache. The physical difference between both architectures can be visualized in Figure 3.2: GPUs dedicate most of their area for processing units (in green), while CPUs dedicate most of their area for execution control and data cache (in yellow and orange, respectively).

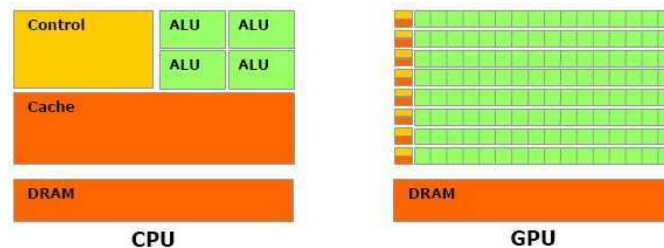


Figure 3.2: Comparison between GPUs and CPUs structures. Source: [NVIDIA, 2014]

A CUDA application consists in code that is executed on CPU and functions (called *kernels*) that are executed on GPU. The GPU is able to do parallel processing by creating threads such that each thread may execute the kernel operations in different data. Thus, the GPU is used as a coprocessor that is able to perform certain tasks more efficiently than the CPU.

3.3 Related Work

There are some important works related to the problem addressed on this paper. In Ben-Moshe [2005], the author presented an algorithm to site facilities using an approach based on radio locator, 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. In Ben-Shimol et al. [2007], the authors described 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 goal of these two works is not to achieve a given coverage of the terrain, they are related to the problem addressed on this paper because they use some important concepts related to our approach.

Other papers describing solutions to site observers on terrains are Franklin and Vogt [2004a,b, 2006]. They are based on the method *Site* proposed in Franklin [2002] which is described in section 3.4 and, as mentioned before, this method uses a greedy strategy to site observers on a terrain. In Magalhães et al. [2010], 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. Additionally, the viewshed representation used in the original method *Site* was improved to require a smaller amount of memory.

In Magalhães et al. [2010]; Magalhães et al. [2011a] were presented two additional methods to site observer on terrains. They are described in section 3.5 and both are based on the method *Site*. In Magalhães et al. [2010], the original method *Site* was extended including a local search heuristic to try to reduce the number of observers selected to achieve the desired coverage. In Magalhães et al. [2011a], some routines used in the method proposed in Magalhães et al. [2010] were implemented in parallel using GPU.

In this paper, we present a more efficient parallel implementation of the method described in Magalhães et al. [2010] that uses a faster implementation (in parallel) of the local search. In section 3.5, we present a better description of the method proposed in this paper and also, the differences with the method presented in Magalhães et al. [2011a].

3.4 The site method

Considering that the observer siting problem is NP-Hard, Franklin [Franklin, 2002; Franklin and Vogt, 2006] proposed an approximate heuristic solution, called *Site*, to find a set of observers to cover the terrain. More precisely, this method uses a greedy approach to obtain a set $\mathcal{S}$ of observers such that a given percentage of the terrain is covered. Initially, $\mathcal{S} = \{\}$ 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 $\mathcal{S}$. As described in Franklin [2002], the details are as follows.

1. Estimate the visibility index of each point in the terrain M. More precisely, determine the points that have a certain minimum visibility index with a certain confidence level. This may be achieved by sampling random targets.

2. Compute $\mathcal{P} = \{P_i\}$ as the set of points with the largest visibility indices. However, do not select two points that are too close together, since their viewsheds will probably overlap considerably, and hence, some of them will be redundant.

3. Compute V_i, the viewshed of each P_i.

4. Initialize $\mathcal{S} = \{\}$. This will accumulate the set of actual observers $\mathcal{S} \subseteq \mathcal{P}$.

5. Initialize $\mathcal{V}$, the joint viewshed of $\mathcal{S}$, that is, the union of the viewsheds of all P_i in $\mathcal{S}$.

6. Repeat the following until a termination condition is satisfied. Typical conditions require $|\mathcal{S}|$ to achieve a certain maximum, or $\mathcal{V}$ to achieve a certain minimum of visible points.

   a) Iterate through $\mathcal{P}$ to find the P_i that will cause the joint visibility index Ω to increase the most. That involves repeatedly counting the number of 1 bits in the union of the joint viewshed $\mathcal{V}$ and V_i.

   b) Insert that P_i into $\mathcal{S}$ and update $\mathcal{V}$.

3.5 Observer siting in GPU

In Magalhães et al. [2010] was presented an extension of the method *Site*, named *Site+*, where some heuristics were included to achieve the same terrain coverage using fewer observers. In order to make these heuristics more efficient, the method *SiteGPU* [Magalhães et al., 2011a] implemented the following operations in GPU: (1) computing the visibility index of viewsheds; (2) finding the candidate observer that will most increase the visibility index of a joint viewshed; and (3) computing the union of viewsheds. In this implementation it was used the following strategy: all data (the viewsheds and joint viewsheds) were kept in the GPU global memory and the heuristics were executed on CPU. Thus, the GPU was used as a coprocessor to efficiently perform operations requested by the heuristics.

Also, to accelerate the viewshed operations, in *SiteGPU* the viewsheds were represented as a small piece of the terrain matrix. More precisely, each viewshed was represented by a $(2R + 1) \times (2R + 1)$ matrix where R is the radius of interest of the observer which is sited on this matrix center. That strategy improves the algorithm efficiency because the points outside the observer's radius of interest are not considered, since they are, by definition, not visible.

The main operation performed by *Site+* is the *swap heuristic* that tries to increase the joint visibility index of the current partial solution without changing the number of observers selected. The basic idea is to check whether swapping a selected observer (in the current partial solution) with another observer didn't

selected yet would increase the joint visibility index. This checking step considers all pairs composed by one observer in the partial solution and another one not in that solution and selects the pair that causes the highest contribution for the joint visibility index.

Notice that increasing the joint visibility index of a partial solution can reduce the number of steps required by the greedy strategy to achieve the final solution and, thus, the required coverage could be achieved using a smaller number of observers (that could be an important improvement since the observer can be an expensive facility, such as a cellular tower). But, on the other hand, this local search performs several viewshed operations and it is often the bottleneck of both *Site+* and *SiteGPU*.

In this paper we propose a more efficient method, named *SiteGSM*, to site observers on terrain. It is based on *Site+* and includes a faster implementation of the local search using dynamic programming and GPU's shared memory which is much faster than the GPU's global memory (used by *SiteGPU*). It is important to mention that both methods *Site+* and *SiteGSM* obtain exactly the same solution, that is, the same number of observers (sited on same terrain places) and, of course, the same terrain coverage. The difference is that *SiteGSM* is much faster than *Site+*.

3.5.1 The Local Search - Swap

Given a set with n candidate observers, let $A = \{V_1, \cdots, V_n\}$ be the set with their corresponding viewsheds, that is, V_i is the viewshed of observer i and let S be a subset of A with k viewsheds representing an initial solution for the observer siting problem. The goal of the swap heuristic is to iteratively change S in order to increase the joint visibility index while keeping constant the number of observers in S.

The local search method is based on the concept of neighborhood of a solution which can be defined as follows: given a solution $S = \{V_{i_1}, \cdots, V_{i_k}\}$, a neighbor of S is a solution S' where an element of S is replaced by another element not in S. See Figure 3.3. In each iteration the current solution is replaced by its best neighbor (the one with highest visibility index).

The process of replacing the current solution with its best neighbor is repeated until it is obtained a solution having no better neighbor, which is a local optimum.

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 is, in fact, correspond to the joint viewshed of the observers whose indices are $i_1, \cdots, i_k$. Thus, the heuristic may be

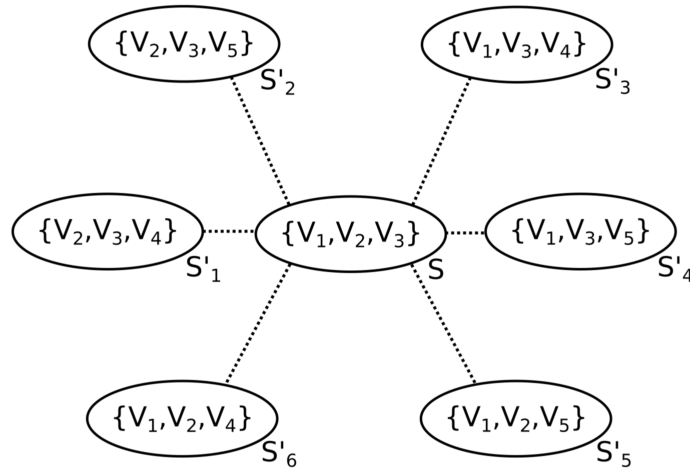


Figure 3.3: Given $A = \{V_1, V_2, V_3, V_4, V_5\}$, S' are the neighbors of the solution $S = \{V_1, V_2, V_3\}$.

implemented (sequentially) as follows: given the set of candidate viewsheds $A = \{V_1, \cdots, V_n\}$, 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}. 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 calculated by computing the number of visible points in $\mathcal{V}_{\overline{r}} \oplus V_j$ 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 2 presents the code for this step that computes the number of visible points in $\mathcal{V}_{\overline{r}} \oplus V_j$ (for all $r = 1 \cdots k$ and $j = 1 \cdots n; j \neq r$) and stores them in the element $Vix[r][j]$. In the next step, this matrix will be used to find the best neighbor of S.

For efficiency improvement, 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.

3.5.2 An Efficient Swap Heuristic Implementation

Notice that, to generate $\mathcal{V}_{\overline{r}}$, for $r = 1 \cdots k$, Algorithm 2 performs $\Theta(k^2)$ union operations $\oplus$, where each union operation involves $\Theta(vsize)$ positions in the viewshed

Algorithm 2 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 matrix Vix, where $Vix[r][j]$ is the joint visibility index of a solution replacing observer r with j.

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

matrices. But, as described below, this step can be improved considerably using dynamic programming.

Given a solution $S = \{i_1, \cdots, i_k\}$, i.e., $V_{i_1} \oplus \cdots \oplus V_{i_k}$, for each $r \in \{1, \cdots, k\}$, we have $\mathcal{V}_{\overline{r}} = (V_{i_1} \oplus \cdots \oplus V_{i_{r-1}}) \oplus (V_{i_{r+1}} \oplus \cdots \oplus V_{i_k})$. Doing $\lambda_r^- = V_{i_1} \oplus \cdots \oplus V_{i_{r-1}}$ and $\lambda_r^+ = V_{i_{r+1}} \oplus \cdots \oplus V_{i_k}$ we can observe 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 3.4 illustrates $\mathcal{V}_{\overline{r}}$ computation for $k = 5$: in this figure, each row r represents the value of $\mathcal{V}_{\overline{r}}$ where the elements in the left of the r-th column represents λ_r^- and in the right, λ_r^+. Notice that λ_4^- can be computed by joining the viewsheds V_{i_1} and V_{i_2} (which is the value of λ_3^-) with viewshed V_{i_3}. On the other hand, the λ^+ values can be computed in similar way using the reverse order.

1		V_{i_2}	V_{i_3}	V_{i_4}	V_{i_5}
2	V_{i_1}		V_{i_3}	V_{i_4}	V_{i_5}
3	V_{i_1}	V_{i_2}		V_{i_4}	V_{i_5}
4	V_{i_1}	V_{i_2}	V_{i_3}		V_{i_5}
5	V_{i_1}	V_{i_2}	V_{i_3}	V_{i_4}	

Figure 3.4: Matrix illustrating $\mathcal{V}_{\overline{r}}$ in a solution with $k = 5$ observers.

Based on these recurrence relations, Algorithm 3 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 2 where are performed $\Theta(k^2)$ viewshed unions.

In *SiteGSM*, the Algorithm 3 was implemented in GPU using the following strategy: the viewsheds were kept in GPU's global memory and, then, the union of viewsheds (loop in lines 3, 9 and 15 of Algorithm 3) 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.

Algorithm 3 Compute the matrix $\mathcal{V}$ that stores $\mathcal{V}_{\overline{r}}$, for $r = 1 \cdots k$ using a dynamic programming strategy.

1: $\mathcal{V}_1[k][vsize] \leftarrow \{\{0\}\}$
2: **for** $r \leftarrow 2$ **to** k **do**
3:   **for** $w \leftarrow 1$ **to** $vsize$ **do**
4:     $\mathcal{V}_1[r][w] \leftarrow \mathcal{V}_1[r-1][w]$ **or** $V[S[r-1]][w]$
5:   **end for**
6: **end for**
7: $\mathcal{V}_2[k][vsize] \leftarrow \{\{0\}\}$
8: **for** $r \leftarrow k-1$ **to** 1 **do**
9:   **for** $w \leftarrow 1$ **to** $vsize$ **do**
10:     $\mathcal{V}_2[r][w] \leftarrow \mathcal{V}_2[r+1][w]$ **or** $V[S[r+1]][w]$
11:   **end for**
12: **end for**
13: $\mathcal{V}[k][vsize] \leftarrow \{\{0\}\}$
14: **for** $r \leftarrow 1$ **to** k **do**
15:   **for** $w \leftarrow 1$ **to** $vsize$ **do**
16:     $\mathcal{V}[r][w] \leftarrow \mathcal{V}_1[r][w]$ **or** $\mathcal{V}_2[r][w]$
17:   **end for**
18: **end for**

After computing $\mathcal{V}_{\overline{r}}$, the next step is to compute the joint visibility index of the

neighbor solutions, as performed by lines 12 to 18 in Algorithm 2. A straightforward implementation of this step in GPU was presented in Magalhães et al. [2011a], 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 performance efficiently because it requires too many accesses to the global memory, which is much slower than other memories such as the shared memory.

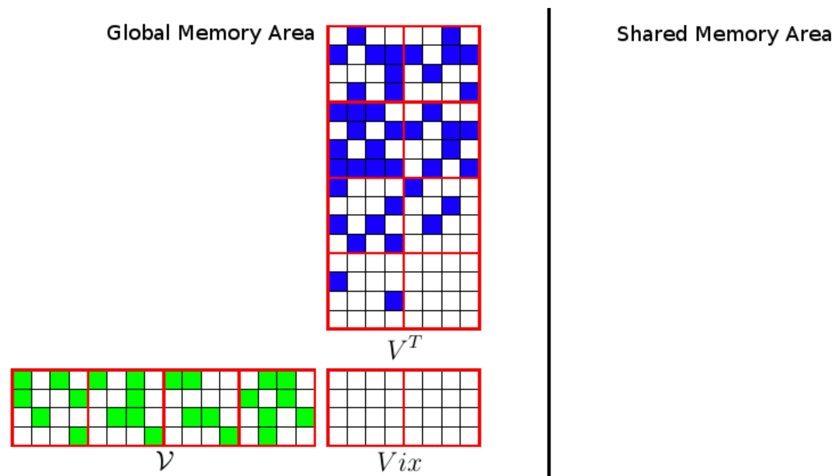
In order to make a better use of the GPU memory hierarchy, we propose a new strategy based on a fast GPU matrix multiplication algorithm. Notice that the joint visibility index, that is, the Vix matrix, is obtained in lines 12 to 18 of Algorithm 2, where the matrices $\mathcal{V}$ and V are overlapped using a bitwise-or operation. In this case, 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 2 can be replaced with

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

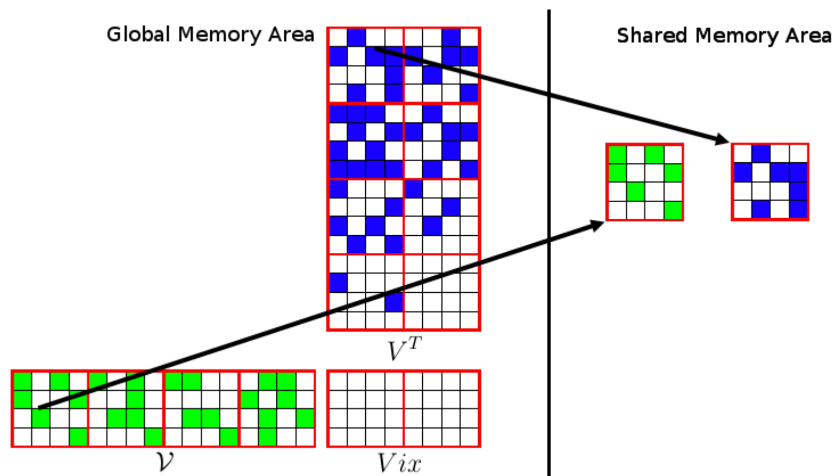
where V^T is the transposed matrix of V.

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 NVIDIA [2014], 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. Figure 3.5 illustrates the algorithm: (3.5a) shows the matrices $\mathcal{V}$, V^T and Vix in the global memory; in (3.5b), one block from $\mathcal{V}$ and one from V^T are loaded in the shared memory and in (3.5c), the resulting block is written back to the global memory. Therefore, most of the algorithm accesses are to the shared memory which is much faster than the global memory.

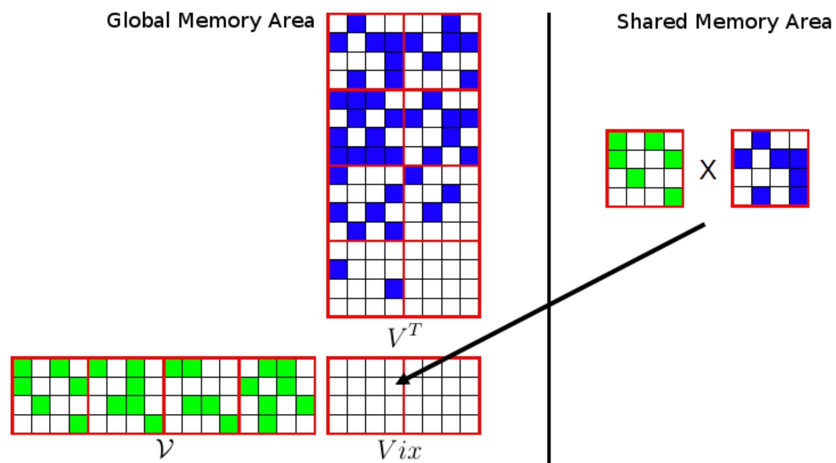
Additionally, since the viewsheds matrices are, usually, sparse (the points outside the observer's radius of interest are always non visible), we adapted the matrix multiplication algorithm to avoid loading and processing matrix blocks where all the elements are 0. A mask matrix was created to label the blocks as following: if all the cells in the block are not visible, the block is labeled with 0; otherwise with 1. See Figure 3.6.



(a) Matrices on global memory.

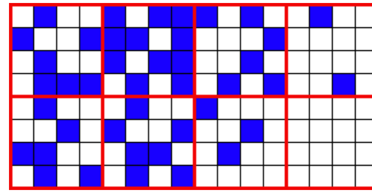


(b) Iterative load in the GPU's shared memory.

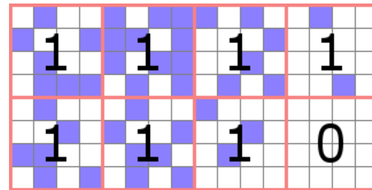


(c) Writing back to the global memory.

Figure 3.5: Matrix Multiplication Algorithm.



(a) Original V matrix.



(b) Mask-matrix over V.

Figure 3.6: The blocks are labeled as following: if all the cells in the block are not visible, the block is labeled with 0; otherwise with 1.

3.6 Experimental Results

The method *SiteGSM* was implemented in C++/CUDA and compiled using *nvcc* 4.0 with maximum optimization level (-O3). It was compared against *SiteGPU* [Magalhães et al., 2011a] and against *Site+* [Magalhães et al., 2010], a sequential CPU version with no use of dynamic programming. The tests were executed on a computer with Dual Intel Xeon E5-2687 3.1GHz, 128GiB of memory running Ubuntu 12.04 LTS and GPU NVidia Tesla Kepler K20x with 6GiB of global memory, 48KB of shared memory per block and 2688 CUDA processing cores and CUDA 5.0.

The tests used different datasets obtained from NASA SRTM webpage: a terrain with 1201×1201 points (90-meter resolution SRTM3 terrain) representing a region of the Minas Gerais state in Brazil and another terrain with 3601×3601 points (30-meter resolution SRTM1 terrain) representing a region of the New Jersey state in USA.

The initial set $\mathcal{P}$ of candidate observers for each terrain was computed using steps 1 to 3 of the *Site* method proposed by Franklin [2002] and described in section 3.4. The *Site* was set up to select 1000 candidates for the smaller terrain and 3000 candidates for the larger one, as in Magalhães et al. [2011a]. Also, the viewsheds of all candidate observers were computed using the same viewshed algorithm used by *Site*, with the observers and target points sited 30 meters above the terrain. This value was chosen because, in general, it is the height of real towers, as for example, communication antennas [Delmellea et al., 2005].

The observer siting methods were tested using radii of interest 100, 200 and

300 points for the first terrain; and 200, 300 and 400 points for the second terrain. For each terrain, the desired coverages (joint visibility indices) were 75%, 85% and 95% of the terrain area. Figure 3.7 shows three examples, with radii of interest 100, 200 and 300 and desired coverage of 25%, indicating how observers were sited on the terrain.

Table 3.1 presents the processing time (in seconds) spent by each method. Column #Obs shows the number of observers sited in each case and the *SiteGSM*'s speedup is shown in parentheses. The symbol * represents a case for which the set $\mathcal{P}$ of candidate observers is not enough to achieve the desired coverage. The symbol ∞ indicates those cases where the processing time is greater than 7 hours.

As the tests showed, in all cases, the method *SiteGSM* proposed in this paper was faster than *SiteGPU* [Magalhães et al., 2011a] - in some cases, more than 20 times - and, as expected, much faster than the *Site+* (more than 1200 times). Notice that, in almost all cases, the higher the desired coverage, the higher the speedup. This can be explained by the fact that higher coverages require more observers and consequently more viewsheds must be processed.

While much faster than *Site+*, *SiteGSM* obtains the same solution as the former and, as presented in Magalhães et al. [2010], the *Site+* can achieve similar terrain coverage as the original *Site* using about 10% less observers. This can be an important (economic) improvement since the observers can represent some expensive facility, as for example, a communication antenna.

3.7 Conclusion

We presented an efficient implementation of a heuristic to site observers on terrains to achieve a given coverage of this terrain. The method is based on an extension of *Site* [Franklin, 2002], where it was included a local search strategy that tries to improve the coverage obtained by a fixed number of observers. This local search was implemented in parallel using graphics processing units and also dynamic programming.

As the tests showed, the method described in this paper is more efficient than a previous method presented in Magalhães et al. [2011a] and, in some cases, more than 20 times faster. Also, it is up to 1200 times faster than the corresponding sequential implementation.

This is an interesting contribution since the extended method *Site+* using local search heuristic can achieve a desired coverage using a smaller number of observers

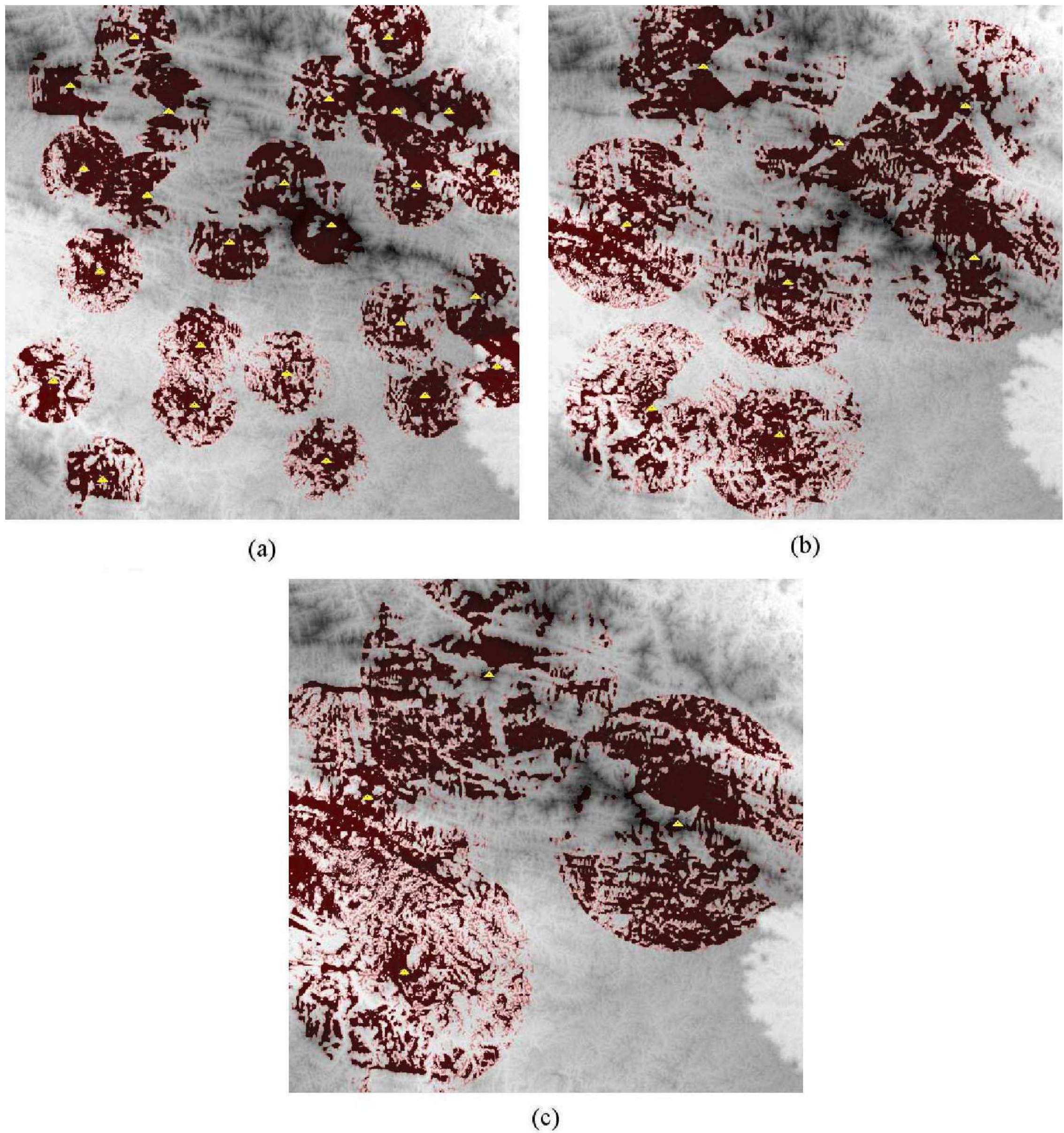


Figure 3.7: Observers sited in terrain 1201x1201 with desired coverage 25% and radii of interest: 100 (a), 200 (b) and 300 (c). The observers are indicated by yellow triangles and visible points are showed in red.

than the original *Site* and thus, better solutions can be generated faster.

As a next step, to improve the proposed method performance even more, we will try to reduce the viewshed matrix size using only a bounding box containing the observer radius of interest. Thus, it will be possible to reduce the volume of data to be processed and, in this case, it seems that we could compute the joint visibility indices adapting a GPU sparse matrix multiplication algorithm.

Table 3.1: Processing time (in seconds) of three methods: two parallel methods using GPU (*SiteGSM* and *SiteGPU*) 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 (in sec.)				
				SiteGSM	*SiteGPU*		*Site+*	
1201 × 1201	100	75%	162	12	180	(15.0)	11010	(917.5)
		85%	299	33	545	(16.5)	∞	(-)
		95%	*	*	*	(-)	*	(-)
	200	75%	55	3	35	(11.7)	1304	(434.7)
		85%	97	6	104	(17.3)	4020	(670.0)
		95%	323	48	888	(18.5)	∞	(-)
	300	75%	34	2	19	(9.5)	479	(239.5)
		85%	62	4	70	(17.5)	1826	(456.5)
		95%	216	28	566	(20.2)	19408	(693.1)
3601 × 3601	200	75%	81	76	422	(5.6)	∞	(-)
		85%	97	110	708	(6.4)	∞	(-)
		95%	125	183	1341	(7.3)	∞	(-)
	300	75%	36	28	160	(5.7)	∞	(-)
		85%	42	41	276	(6.7)	∞	(-)
		95%	54	66	523	(7.9)	∞	(-)
	400	75%	20	14	81	(5.8)	14867	(1061.9)
		85%	24	19	124	(6.5)	22869	(1203.6)
		95%	30	30	270	(9.0)	∞	(-)

Table 3.2: 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 $V_{i_{r+1}} \cdots V_{i_k}$

4. An efficient GPU multiple-observer siting method based on sparse-matrix multiplication[1]

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 very 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. 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, as the tests showed, the new method is much more efficient than the previous ones, mainly because it can process much larger terrains (the older methods are restrictive about terrain size) and, also, it is faster.

4.1 Introduction

Terrain modeling plays an important role in geographical information science (GIS) and many applications concern visibility, that is, determining the set of points that are visible from a particular point, called observer, which can be located at some height above the terrain. These applications include telecommunications, environmental planning, autonomous vehicle navigation and military monitoring [Franklin

[1]Este capítulo apresenta o artigo "*An efficient GPU multiple-observer siting method based on sparse-matrix multiplication*", submetido à ACM SIGSPATIAL 2014 (*22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*) [Pena et al., 2014b].

and Ray, 1994; Li et al., 2005; Nagy, 1994; Andrade et al., 2011]. 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 [Hrovat et al., 2010], or monitoring cameras or towers [Ben-Moshe, 2005; Ben-Shimol et al., 2007].

As described in Nagy [1994], the siting problem is NP-Hard and, therefore, there is no known efficient algorithm to find its optimal solution. Thus, in general, it is used a heuristic 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 geographical 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.

In this paper we present an approximate solution for an instance of the multiple observers 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 Franklin [2002], which was based on a greedy strategy and, 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 Graphic Processing Unit (GPU) and dynamic programming.

This local search strategy for reducing the number of observers was already used in Magalhães et al. [2011a]; Pena et al. [2014a] 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).

But both methods described in those papers are very restricted since they cannot process large terrains. As presented in the papers, the largest terrains that these methods can process have about 3601×3601 cells. And, as mentioned above, the 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) and 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 with the other methods).

4.2 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 point of the earth surface 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*. As usual, 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.

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 4.1. In this Figure, T_1 is visible from O but T_2 is not.

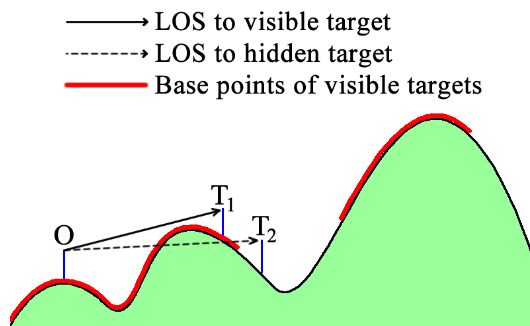


Figure 4.1: Visibility queries using a line of sight. Source: [Magalhães et al., 2011b]

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 area of the terrain that is seen by at least one observer [Franklin and Ray, 1994].

The *joint viewshed* 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 [Nagy, 1994] 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 terrain.

4.3 Related Work

There are some important related works addressing the siting problem. In Ben-Moshe [2005], the author presented an algorithm to site facilities using an approach based on radio locator, 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. In Ben-Shimol et al. [2007], the authors described 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 from 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 [2002] 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 Magalhães et al. [2010]; Magalhães et al. [2010]; Magalhães et al. [2011a]; Pena et al. [2014a]. In Magalhães et al. [2010] 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. Additionally, the viewshed representation used in the original 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 Magalhães et al. [2010]; Magalhães et al. [2011a]; Pena et al. [2014a], 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, this new strategy may achieve a feasible solution in less iterations (that is, using less observers).

These three methods differ from each other mainly in how the local search is implemented. In Magalhães et al. [2010], the local search was implemented using sequential programming running in CPU. In Magalhães et al. [2011a], a first parallel version of the local search was implemented using GPU. And, as the amount of data being processed is too big, these data are stored and processed in the GPU's global memory (that is the biggest, but the slowest memory in the GPU's memory hierarchy). In Pena et al. [2014a] 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. 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 will always obtain the same solution, that is, select the same set of observers because they are based on the same heuristic. But, although the two methods *SiteGPU* and *SiteGSM* are efficient (considering their execution time)[2], 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 often 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.

4.4 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 Pena et al. [2014a]. 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, even though 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.

4.4.1 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

[2]In fact, as described in Pena et al. [2014a], *SiteGSM* is more than 20 times faster than *SiteGPU*.

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.

Thus, 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 4.2 presents an example of the solutions that are evaluated performing one iteration of the swap local search in the solution $S = \{V_1, V_2, V_3\}$.

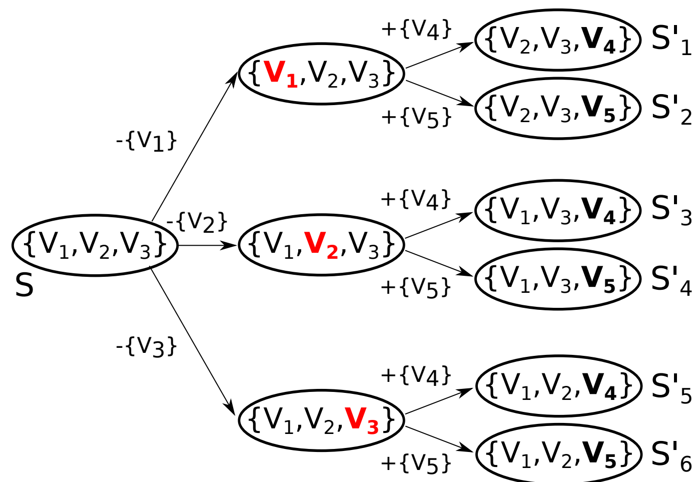


Figure 4.2: Given $V = \{V_1, V_2, V_3, V_4, V_5\}$, each S'_i is obtained performing a swap in $S = \{V_1, V_2, V_3\}$.

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 observers 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 $j \neq i_r$.

The most time-consuming step in this heuristic is computing the joint visibility index for each neighbor solution. Algorithm 4 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 $j = 1 \cdots n; j \neq i_r$) and stores it in the element $Vix[j][r]$. In the next step, this matrix will be used to find the best neighbor of S.

Algorithm 4 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 $S[r]$ with j.

1: $Vix[n][k] \leftarrow \{\{0\}\}$
2: $\mathcal{V}[k][VSize] \leftarrow \{\{0\}\}$
3: **for** $r \leftarrow 1$**to** k **do**
4:    **for** $m \leftarrow 1$**to** k **do**
5:       **if** $r \neq m$ **then**
6:          **for** $w \leftarrow 1$**to** *VSize* **do**
7:             $\mathcal{V}[r][w] \leftarrow \mathcal{V}[r][w]$ **or** $V[S[m]][w]$
8:          **end for**
9:       **end if**
10:    **end for**
11: **end for**
12: **for** $r \leftarrow 1$**to** k **do**
13:    **for** $j \leftarrow 1$**to** n **do**
14:       **for** $w \leftarrow 1$**to** *VSize* **do**
15:          $Vix[j][r] \leftarrow Vix[j][r] + (V[j][w]$ **or** $\mathcal{V}[r][w]$)
16:       **end for**
17:    **end for**
18: **end for**
19: *return* Vix

For efficiency improvement, as in the previous works [Magalhães et al., 2010; Magalhães et al., 2011a; Pena et al., 2014a], 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.

4.4.2 An Efficient Swap Heuristic Implementation

Observe that Algorithm 4 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 4.3 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.

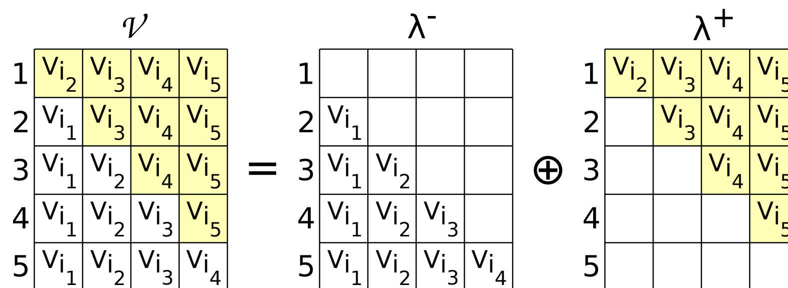


Figure 4.3: Matrix illustrating $\mathcal{V}_{\overline{r}}$ in a solution with $k = 5$ observers.

Based on these recurrence relations, Algorithm 5 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 4 where are performed $\Theta(k^2)$ viewshed unions.

As presented in *SiteGSM* method proposed by Pena et al. [2014a], the $\mathcal{V}$ computation given in Algorithm 5 can be implemented in GPU using the following strategy: the viewsheds were kept in GPU's global memory and, then, the union of

viewsheds (loop in lines 3, 9 and 15 of Algorithm 5) 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.

Algorithm 5 Compute the matrix $\mathcal{V}$ that stores $\mathcal{V}_{\overline{r}}$, for $r = 1 \cdots k$ using a dynamic programming strategy.

1: $\mathcal{V}_1[k][VSize] \leftarrow \{\{0\}\}$
2: **for** $r \leftarrow 2$ **to** k **do**
3: **for** $w \leftarrow 1$ **to** $VSize$ **do**
4: $\mathcal{V}_1[r][w] \leftarrow \mathcal{V}_1[r-1][w]$ **or** $V[S[r-1]][w]$
5: **end for**
6: **end for**
7: $\mathcal{V}_2[k][VSize] \leftarrow \{\{0\}\}$
8: **for** $r \leftarrow k-1$ **to** 1 **do**
9: **for** $w \leftarrow 1$ **to** $VSize$ **do**
10: $\mathcal{V}_2[r][w] \leftarrow \mathcal{V}_2[r+1][w]$ **or** $V[S[r+1]][w]$
11: **end for**
12: **end for**
13: $\mathcal{V}[k][VSize] \leftarrow \{\{0\}\}$
14: **for** $r \leftarrow 1$ **to** k **do**
15: **for** $w \leftarrow 1$ **to** $VSize$ **do**
16: $\mathcal{V}[r][w] \leftarrow \mathcal{V}_1[r][w]$ **or** $\mathcal{V}_2[r][w]$
17: **end for**
18: **end for**

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 4. A straightforward implementation of this step in GPU was presented in Magalhães et al. [2011a], 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 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* [Pena et al., 2014a] 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 4). 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 4 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 NVIDIA [2014], 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.

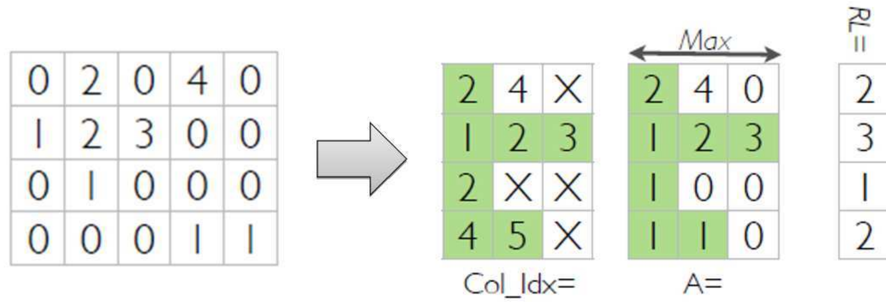
4.4.3 Using a sparse-dense matrix multiplication algorithm

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[3]) 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.

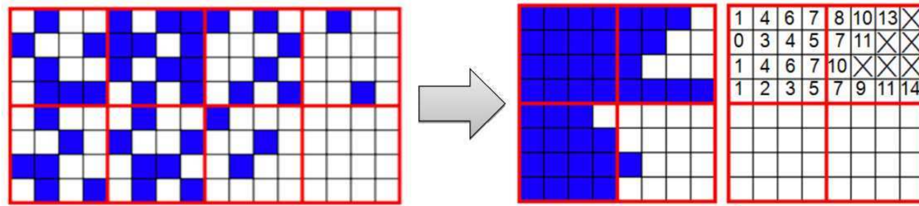
In this work, the matrix V was coded using ELLPACK-R matrix format [Ortega et al., 2013]. Figure 4.4 shows a simple matrix coded using ELLPACK-R and the illustration of a viewshed matrix coded using this format. Considering this representation, the overlay of the viewsheds in V with the joint viewsheds in $\mathcal{V}$ was performed using an adaptation of the sparse-dense matrix multiplication algorithm proposed by Ortega et al. [2013].

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, replacing the multiplication operation in the scalar product by 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.

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



(a) Source: [Wafai et al., 2009]



(b)

Figure 4.4: ELLPACK-R matrix format representation: (a) an example showing how the matrix is packed; (b) how the viewshed matrix (from Figure 3.6a) is packed using this representation.

However, since the sparse-dense matrix multiplication algorithm proposed in Ortega et al. [2013] does not process 0 in the first (sparse) matrix then this straightforward adaption 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 the one with the largest visible area, we compute the area increment[4] 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

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

operation ((a **or** b) **and** $\sim a$) followed by a bitwise population count.

4.4.4 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 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 (S_r, V_j) such that the visual coverage of S would increase by replacing S_r with V_j. This process checks all S_r with $r = 1 \cdots k$ and all V_j with $j = 1 \cdots n$. In this paper, we present an implementation for this strategy that is not only time efficient, but can also 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 scientific 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 $\mathcal{V}_{\bar{i}} \cdots \mathcal{V}_{\bar{j}}$. These overlays are still computed as described in section 4.4.3, 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 4.4.2 needs to be adapted since, to compute $\mathcal{V}_{\bar{k}}$, 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 4.5. 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 $S_j, S_{j+2}, \cdots, S_k$.

$\mathcal{V}_{3,6}$

3	Vi_1	Vi_2	Vi_4	Vi_5	Vi_6	Vi_7
4	Vi_1	Vi_2	Vi_3	Vi_5	Vi_6	Vi_7
5	Vi_1	Vi_2	Vi_3	Vi_4	Vi_6	Vi_7
6	Vi_1	Vi_2	Vi_3	Vi_4	Vi_5	Vi_7

=

λ^-

3	Vi_1	Vi_2				
4	Vi_1	Vi_2	Vi_3			
5	Vi_1	Vi_2	Vi_3	Vi_4		
6	Vi_1	Vi_2	Vi_3	Vi_4	Vi_5	

$\oplus$

λ^+

3			Vi_4	Vi_5	Vi_6	Vi_7
4				Vi_5	Vi_6	Vi_7
5					Vi_6	Vi_7
6						Vi_7

Figure 4.5: Matrix illustrating $\mathcal{V}_{\overline{3\cdots6}}$ 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. Thus, 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. Observe that 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 4.4.2, 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 GPU.

4.5 Experimental Results

In order to evaluate the proposed method, we defined a set of tests and compared it against *Site+* [Magalhães et al., 2010], that is a sequential implementation of the *swap* heuristic and against *SiteGSM* [Pena et al., 2014a], 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 Magalhães et al. [2011a] was not evaluated in our tests because, as shown in Pena et al. [2014a], *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 and CUDA 5.0.

The tests were performed in 4 terrains containing 30-meter-resolution elevation data from the state of Illinois, in the United States, and were obtained from the NASA SRTM project [Rabus et al., 2003]. The terrains, that are numbered from 1 to 4, represent square regions containing respectively 1201^2, 3601^2, 7500^2 and 15000^2 cells.

Initially, the set $\mathcal{P}$ of candidate observers was generated using the observer selection step of the *Site* method [Franklin, 2002], that selects observers with high visibility indices in the terrain. The number of candidate observers 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. For the viewshed computation, the observers were positioned 30 meters above the terrain since this value represents typical elevation for communication antennas [Delmellea et al., 2005].

Since the main contribution of *SparseSite* is its ability to efficiently process high resolution terrains, in our first set of tests we evaluated its efficiency to process large terrains. The results are presented in Table 4.1. In these tests, it was not possible to compare *SparseSite* against other methods because *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 4.4.4, *SparseSite* can be configured to use different sizes for the portion of the dense matrix that is kept in the GPU's memory. Thus, in exchange of 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 the *SparseSite* performance versus the amount of memory available for the heuristic, we performed a set of tests considering different values for n_r (that defines the number of rows of the dense matrix kept in the GPU's memory). Table 4.2 and chart in Figure 4.6 present the running time (in seconds) and total memory usage (in MB) of *SparseSite* as a function of n_r. These tests were performed in Terrain 3 using radius of interest with 200 cells and desired coverage of 95%.

Table 4.1: Processing time of *SparseSite* in large terrains.

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

Table 4.2: Processing-time and memory usage of *SparseSite* as a function of the number of rows (n_r) of the dense matrix that is kept in the memory. These tests were performed on Terrain 3, using radius of interest equal to 200 cells and desired 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

Observe that, except for the case where only one line of the dense matrix is kept in the memory, there is a small variation in the heuristic performance. In fact, the difference between the test case where $n_r = 5$ and $n_r = 260$ (the largest number of rows that can be managed in 6GB of memory of K20x GPU) is smaller than 5%. This is an important result indicating that, differently of the previous methods, the proposed heuristic can process big terrains even in lower ends GPUs.

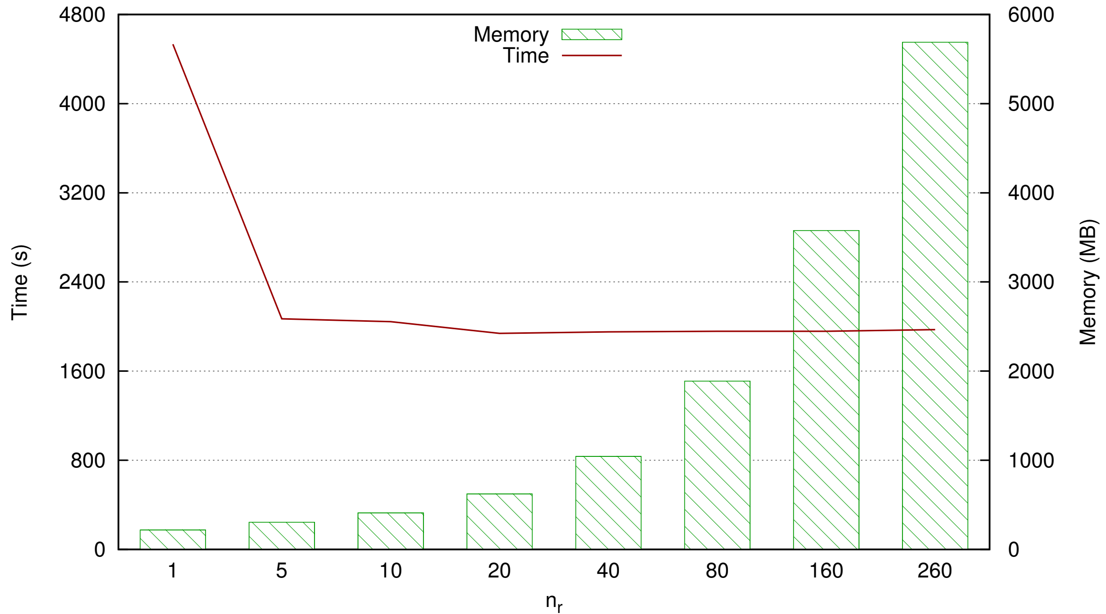


Figure 4.6: Processing-time and memory usage of *SparseSite* as a function of the number of rows (n_r) of the dense matrix that is kept in the memory.

To compare the *SparseSite* processing time against other methods, we performed a set of tests in small terrains that can be processed in a feasible amount of time by the sequential implementation of the swap heuristic (*Site+* [Magalhães et al., 2010]) and, also, that can be processed using less than the available 6GB of memory by the previous parallel implementation (*SiteGSM* [Pena et al., 2014a]). 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 4.3 and charts in Figure 4.7. Column #Obs shows the number of observers sited in each case and the speedup of each method (when 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 times, while the greatest speedup obtained by *SiteGSM* was 2762 times. Also, in most cases, *SparseSite* was faster than *SiteGSM*, being up to 2.7 times faster. However, *SiteGSM* was faster in situations where the radius of interest was larger. This behaviour can be explained because in these cases the number of observers needed to achieve the desired coverage is very small (for example, in the test case in Terrain 1 with radius of interest equal to 300 cells and desired coverage of 75%, the algorithm used only

Table 4.3: 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). The speedup of each method (compared against *Site+*) is shown in parenthesis.

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

4 observers) and, thus, the overheads associated with the sparse-matrix processing techniques made *SparseSite* slower.

As shown by Magalhães et al. [2010], the use of local search procedures with greedy heuristics can generate solutions that use less observers than the solutions obtained by methods that use only a 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 4.4 presents a comparison between *SparseSite* (the greedy method improved with the proposed swap heuristic) and by the *Site* heuristic (that uses only a greedy method). These tests were performed in terrains 2, 3 and 4 using radius of interest 400 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 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.

Table 4.4: Comparison of the processing time and number of observers used by the *SparseSite* heuristic (greedy with local search) and by the *Site* method (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%

Figures 4.8a and 4.8b compares, respectively, a solution obtained by the greedy method against a solution obtained using the greedy method with the *swap* heuristic. This siting was performed in Terrain 1, using radius of interest of 200 cells the desired coverage was 75%. In this figure, we used a small terrain with high radius of interest and low desired coverage in order to point out the difference between the methods. Notice that the solution obtained using the *swap* heuristic used 9 observers, while the solution obtained using only the greedy method used 10 observers to reach the same minimum coverage.

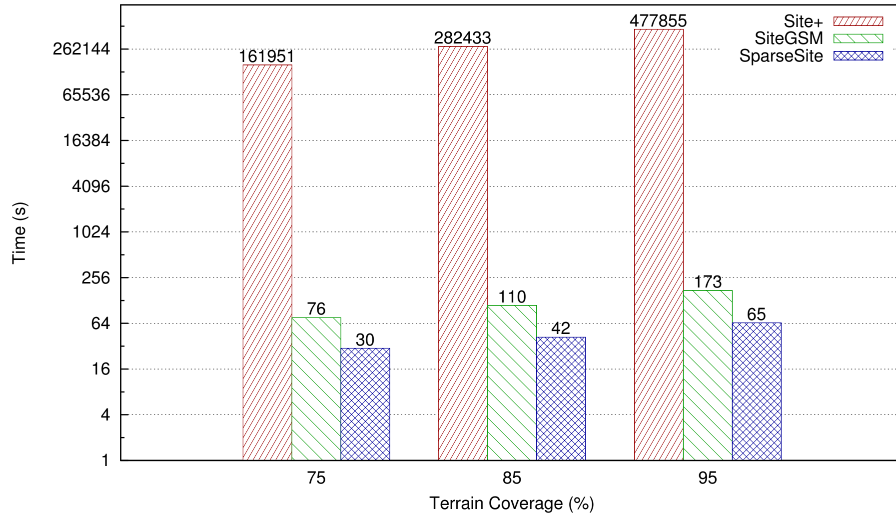
4.6 Conclusions and future work

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

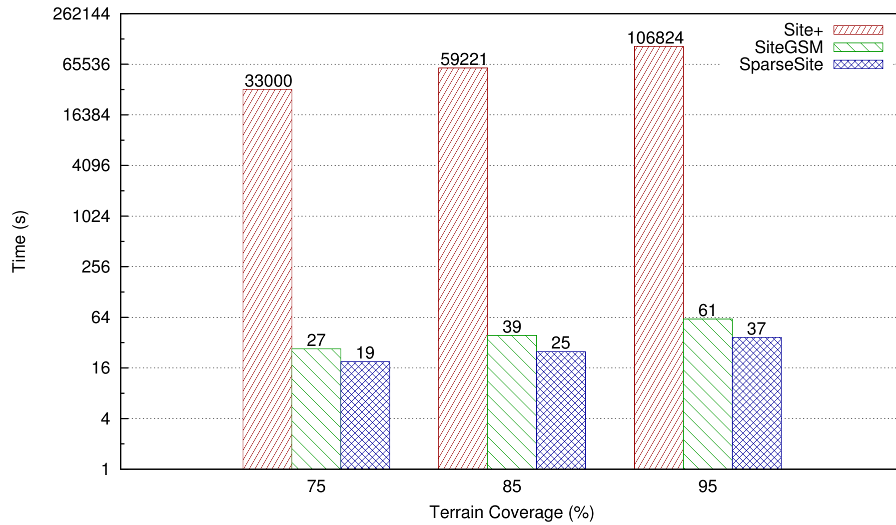
that can be processed by other methods showed that the use of processing techniques specially designed for sparse matrices led to a speedup of up to 2.7 times if compared against the fastest previous parallel implementation and up to 7352 if compared against the sequential method.

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

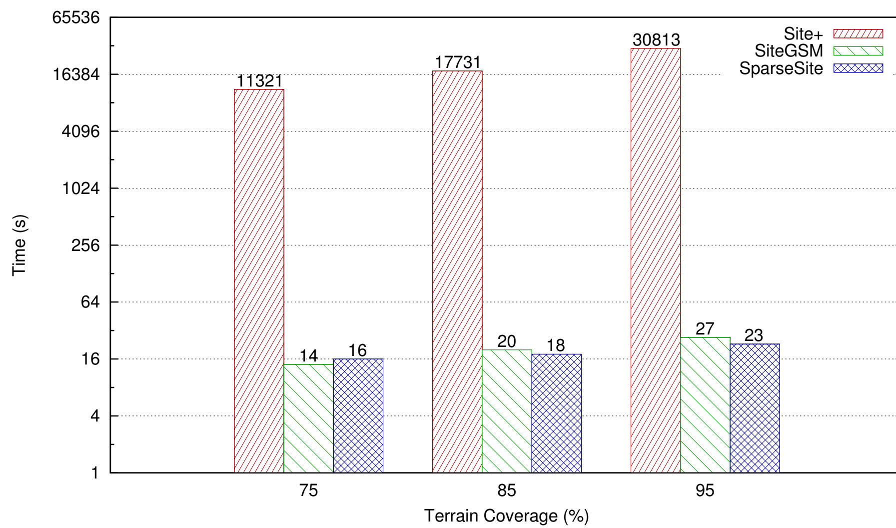
As future work, we intend to improve the performance and memory usage of the proposed implementation for situations where the viewshed data is not sparse (for example, when the observers have radii of interest closer to the terrain size) and, also, to perform experiments using the proposed local search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



(a) $R = 200$.



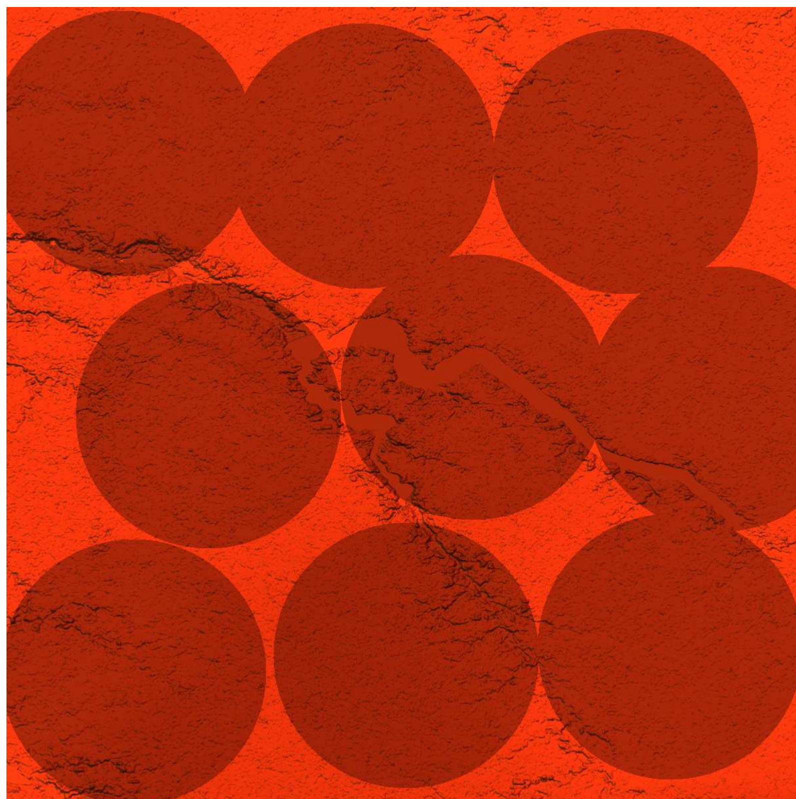
(b) $R = 300$.



(c) $R = 400$.

Figure 4.7: Processing time (in seconds) of the three methods (*SparseSite*, *SiteGSM* and *Site+*) to site observers with radius of interest (R) of 200 (a), 300 (b) and 400 (c) cells on the terrain with 3601^2 cells. For clarity purposes, the y-scale is logarithmic.

(a) Greedy method.



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

Figure 4.8: Comparison of a solution obtained by the greedy method against a solution obtained for the greedy method improved with the *swap* local search.

5. Conclusões gerais e trabalhos futuros

Neste trabalho foi apresentado o algoritmo *SparseSite* para solucionar o problema do posicionamento de observadores em terrenos. Conforme descrito, o *SparseSite* é uma heurística que utiliza uma estratégia gulosa que inclui uma busca local baseada em trocas e ele foi projetado levando em consideração arquiteturas paralelas de placas gráficas (GPUs). Este método é o resultado da evolução do nosso método anterior *SiteGSM*, que foi descrito nos capítulos 2 e 3. Como os resultados experimentais apresentaram, o *SparseSite* se mostrou mais eficiente dos que os algoritmos propostos anteriormente na literatura, *Site+* e *SiteGPU*. Os ganhos (*speedups*) obtidos por ele podem ser importantes para usuários que desejem avaliar aplicações tanto em terrenos de baixa resolução representando maiores regiões, quanto em dados mais precisos com terrenos de alta resolução.

No capítulo 2 foi apresentado a primeira versão do *SiteGSM* que é uma extensão do método *Site* [Franklin, 2002] e inclui uma busca local para tentar melhorar a cobertura de uma sub-solução sem alterar o número de observadores dessa sub-solução. Esta primeira versão foi comparada com o método *Site+* [Magalhães et al., 2010] e foi cerca de 200 vezes mais eficiente na maioria dos casos de teste avaliados.

No capítulo 3, foram incluídas duas melhorias importantes no método *SiteGSM* em ambas etapas de processamento. Na etapa de programação dinâmica, as sobreposições entre *viewsheds*, antes realizadas em CPU, passaram a ser realizadas em GPU e na etapa que determina a área visível de cada *viewshed* candidato, uma máscara foi criada sobre a matriz de candidatos para reduzir o número de operações que estavam sendo realizadas na GPU sobre dados da entrada que não influenciavam no resultado. Esta segunda versão do *SiteGSM* foi comparada com os métodos *Site+* e *SiteGPU* [Magalhães et al., 2011a] e os resultados mostraram que o novo algoritmo *SiteGSM* chegou a ser, em alguns casos, mais de 20 vezes mais rápido do que o método *SiteGPU* e mais de 1200 vezes mais rápido do que o método *Site+*.

No capítulo 4 foi apresentado o método *SparseSite* que é uma nova implementação do nosso método anterior *SiteGSM*, onde foram incluídas algumas melhorias que permitiram que maiores volumes de dados pudessem ser processados. As melhorias incluem o uso de matrizes esparsas para representar o conjunto de *viewsheds* candidatos que é considerado esparso e na adaptação de um algoritmo para multiplicação destas matrizes realizando sobreposição de matrizes além de uma contagem

para determinar a área total de visibilidade de cada candidato. Além disso, para reduzir a quantidade de memória usada pelo método, foi adotada a estratégia de particionar o processamento da matriz que armazena a sobreposição dos *viewsheds* de uma sub-solução. Este algoritmo foi comparado com os métodos *Site+* e *SiteGSM*. O *SparseSite* chegou a ser, em alguns casos, quase 3 vezes mais rápido do que o método *SiteGSM* gastando apenas 10% da memória usada por ele e mais de 7000 vezes mais rápido do que o método *Site+*. Além disso, foram realizados testes do *SparseSite* em maiores volumes de dados que obtiveram resultados em tempos razoáveis. Esses testes não puderam ser realizados nos outros métodos avaliados devido à limitações de memória ou por demandarem muito tempo de processamento.

A eficiência obtida pelo algoritmo *SiteGSM* ocorreu por dois fatores. O primeiro deles é o uso da estratégia de programação dinâmica que reduz a complexidade da etapa de um tempo quadrático para um tempo linear. O segundo fator é que a maioria dos acessos aos dados são feitos na memória compartilhada da GPU, que comparada à memória global, é muito mais rápida. Isso se deve à adaptação do algoritmo de multiplicação de matrizes. No caso do método *SparseSite*, a eficiência obtida ocorreu porque o algoritmo aproveita melhor o fato da matriz ser esparsa e, com isso, processa muito menos dados. Isso fica ainda mais evidente em casos de testes como os avaliados no capítulo 4, onde o tamanho do raio dos observadores é muito menor do que a dimensão dos terrenos.

Por fim, como demonstrado no Capítulo 4, o método apresentado neste trabalho atinge, de forma eficiente, a mesma taxa de cobertura do que o método que usa apenas a estratégia gulosa e usando um número menor de observadores. Do ponto de vista prático isso é uma grande contribuição visto que os observadores podem representar instalações de custo elevado, como torres de telefonia móvel por exemplo. Além disso, do ponto de vista teórico, os avanços adquiridos neste trabalho podem servir de base para a construção de novos algoritmos paralelos voltados para outras aplicações.

Como trabalhos futuros pretende-se analisar situações em que os dados não são muito esparsos, isto é, quando o número de pontos nos *viewsheds* dos observadores é próximo ao tamanho do terreno avaliado. Nesse caso pode-se adotar estratégias de processamento externo onde os dados ficariam armazenos em disco e divididos em blocos para serem processados por partes pelas estratégias de programação paralela. Além disso, pretende-se avaliar a busca local proposta neste trabalho em outras heurísticas que fazem uso desse tipo de estratégia como *GRASP* e *Simulated Annealing*.

Referências Bibliográficas

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

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

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

Bespamyatnikh, S., Chen, Z., Wang, K., and Zhu, B. (2001). On the planar two-watchtower problem. In *7th International Computing and Combinatorics Conference*, pages 121--130. Springer-Verlag London.

Câmara, G., Davis, C., and Monteiro, A. M. (2001). *Introdução à Ciência da Geoinformação*. Instituto de Pesquisa Espacial- INPE, São Jose dos Campos, SP, Brasil, Disponivel em: http://www.dpi.inpe.br/gilberto/livro/introd/ — Acessado em Junho 2014.

Delmellea, E. M., Rogersonb, P. A., Akellad, M. R., Battae, R., Blattf, A., and Wilsonf, G. (2005). A spatial model of received signal strength indicator values for automated collision notification technology. *Transportation Research Part C: Emerging Technologies*, 13(5-6):432--447.

Eidenbenz, S. (2002). Approximation algorithms for terrain guarding. *Information Processing Letters*, 82(2):99--105.

Felgueiras, C. A. (2001). Modelagem numérica de terreno. In G. Câmara, C. Davis, A. M. V. M., editor, *Introdução à Ciência da Geoinformação*, volume 1. INPE.

Floriani, L. D., Puppo, E., and Magillo, P. (1999). Applications of computational geometry to geographic information systems. In J. R. Sack, J. U., editor, *Handbook of Computational Geometry*, pages 303--311. Elsevier Science.

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

Franklin, W. R. and Ray, C. (1994). 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.

Franklin, W. R. and Vogt, C. (2004a). Efficient multiple observer siting on large terrain cells. *GIScience 2004*.

Franklin, W. R. and Vogt, C. (2004b). Multiple observer siting on terrain with intervisibility or lo-res data. In *XXth Congress, International Society for Photogrammetry and Remote Sensing*, Istanbul.

Franklin, W. R. and Vogt, C. (2006). Tradeoffs when multiple observer siting on large terrain cells. In Riedl, A., Kainz, W., and Elmes, G., editors, *Progress in Spatial Data Handling: 12th international symposium on spatial data handling*, pages 845--861, Vienna. Springer. ISBN 978-3-540-35588-5.

Goodchild, M. F., Longley, P. A., Maguire, D. J., and Rhind, D. W. (2005). *Geographic information systems and science*, volume 2. John Wiley and Sons, Chichester.

Hrovat, A., Ozimek, I., Vilhar, A., Celcer, T., Saje, I., and Javornik, T. (2010). Radio coverage calculations of terrestrial wireless networks using an open-source grass system. *WSEAS Transactions on Communications*, 9(10):646--657.

Kumler, M. P. (1994). An intensive comparison of triangulated irregular networks (tins) and digital elevation models (dems). *Cartographica: The International Journal for Geographic Information and Geovisualization*, 31(2):1--99.

Laurini, R. and Thompson, D. (1992). *Fundamentals of Spatial Information Systems*. Academic Press.

Li, Z., Zhu, Q., and Gold, C. (2005). *Digital Terrain Modeling — principles and methodology*. CRC Press.

Magalhães, S. V. G., Andrade, M. V. A., and Ferreira, C. R. (2010). Heuristics to site observers in a terrain represented by a digital elevation matrix. In *Anais do XI Simposio Brasileiro de Geoinformatica*, pages 110--121, Campos do Jordão, Brazil.

Magalhães, S. V. G., Andrade, M. V. A., and Ferreira, R. S. (2011a). Using gpu to accelerate heuristics to site observers in dem terrains. In *IADIS Applied Computing (AC 2011)*, pages 127--133. Rio de Janeiro.

Magalhães, S. V. G., Andrade, M. V. A., and Franklin, W. R. (2010). 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.

Magalhães, S. V. G., Andrade, M. V. A., and Franklin, W. R. (2011b). Multiple observer siting in huge terrains stored in external memory. *International Journal of Computer Information Systems and Industrial Management (IJCISIM)*, 3:143--149.

Nagy, G. (1994). Terrain visibility. *Computers & graphics*, 18(6):763--773.

NVIDIA (2014). CUDA C programming guide. *NVIDIA Corporation, July.* Disponivel em: http://docs.nvidia.com/cuda — Acessado em Julho 2014.

Ortega, G., Vázquez, F., García, I., and Garzón, E. M. (2013). Fastspmm: An efficient library for sparse matrix matrix product on gpus. *The Computer Journal*, page bxt038.

Pena, G. C., Magalhães, S. V., Andrade, M. V., and Ferreira, C. R. (2013). Algoritmo paralelo usando gpu para o posicionamento de observadores em terrenos. In *Anais do XIV Simposio Brasileiro de Geoinformatica*, Campos do Jordão, Brazil.

Pena, G. C., Magalhães, S. V., Andrade, M. V., Franklin, W., and Ferreira, C. R. (2014a). An improved parallel algorithm using gpu for siting observers on terrain. In *Proceedings of 16th International Conference on Enterprise Information Systems (ICEIS)*, pages 367--375, Lisboa, Portugal. SCITEPRESS Digital Library.

Pena, G. C., Magalhães, S. V., Andrade, M. V., Franklin, W., Ferreira, C. R., Li, W., and Benedetti, D. (2014b). An efficient gpu multiple-observer siting method based on sparse-matrix multiplication. *Submitted for publication.* Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems(ACM SIGSPATIAL 2014).

Rabus, B., Eineder, M., Roth, A., and Bamler, R. (2003). *The Shuttle Radar Topography Mission (SRTM).* http://www2.jpl.nasa.gov/srtm/ — Acessado em Janeiro 2014.

Wafai, M. et al. (2009). Sparse matrix vector multiplications on graphic processors.

Young-Hoon Kim, Rana, S., and Wise, S. (2004). Exploring multiple viewshed analysis using terrain features and optimisation techniques. *Computers & Geosciences*, 30:1019--1032.