Algoritmo paralelo usando GPU para o posicionamento de observadores em terrenos

Guilherme C. Pena[1], **Salles V. G. Magalhães**[1], **Marcus V. A. Andrade**[1], **Chaulio R. Ferreira**[1]

[1]Departamento de Informática - Universidade Federal de Viçosa (UFV)
Campus da UFV – 36.570-000 – Viçosa – MG – Brasil

{guilherme.pena,salles,marcus,chaulio.ferreira}@ufv.br

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 programmming).*

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 esse método chega a ser acima de 200 vezes mais rápido que a implementação convencional (sem o uso de programação dinâmica e paralela).*

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*). Muitas aplicações práticas, tais como telecomunicações, planejamento ambiental, e monitoramento militar, envolvem esse conceito [Ben-Moshe et al. 2007, Bespamyatnikh et al. 2001]. Um problema importante relacionado à visibilidade é o posicionamento de observadores para maximizar a "cobertura do terreno", sendo que esses "observadores" podem ser câmeras, torres de vigilância ou de telefonia móvel, etc. [Ben-Moshe et al. 2007, Franklin and Vogt 2006, Eidenbenz 2002]. Conforme demonstrado em [Nagy 1994] esse problema é NP-Completo e, portanto, não se conhece um algoritmo eficiente para resolvê-lo de maneira exata.

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

Este trabalho apresenta uma implementação paralela utilizando *Graphics Processing Units (GPUs)* de uma heurística para solucionar o problema do posicionamento de

observadores em um terreno representado por um Modelo Digital de Elevação (MDE) *raster*. Mais precisamente, a heurística proposta visa maximizar a área do terreno coberta por um dado número de observadores.

2. Referencial Teórico

2.1. Visibilidade - Definições

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 mapa de visibilidade (*viewshed*) acumulado desses observadores [Young-Hoon Kim et al. 2004, Magalhães et al. 2010].

2.2. CUDA e programação paralela de propósito geral

Atualmente, uma técnica de programação paralela que tem sido muito utilizada é a *GPGPU* [CUDA Programming Guide 2007], que consiste em se utilizar os vários núcleos de processamento presentes em *GPUs* para processar dados. O modelo de programação *Compute Unified Device Architecture (CUDA)* facilita o desenvolvimento de aplicativos capazes de aproveitar o potencial de processamento das *GPUs* da NVIDIA, que são compostas por vários processadores do tipo *Single Instruction, Multiple Data (SIMD)* e possibilitam a execução de milhares de *threads* de forma paralela.

A arquitetura *CUDA* permite a execução de código tanto no processador principal (*CPU*) quanto na *GPU*. Assim, o *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*.

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 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 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. 2011] 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.

3.1. Heurística de 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 idéia 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*[1]. 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.

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

```
1: Area[1..k][1..n] ← {{0}}
2: for r = 1 → k
3:    for j = 1 → n
4:       for w = 1 → VSize
5:          Area[r][j] ← Area[r][j] + (Υ[r][w]  or  V[j][w])
```

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).

[1] 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.

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

3.2. Implementação eficiente da heurística *Swap*

Note que, para se gerar os valores de Υ_r com $r = 1 \cdots k$, o algoritmo descrito na seção 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 $\overline{\lambda}_r = (\nu_{r+1} \oplus \nu_{i+2} \oplus \cdots \oplus \nu_k) = (\nu_{r+1} \oplus \overline{\lambda}_{r+1})$. Dai, $\Upsilon_r = \lambda_r \oplus \overline{\lambda}_r$. 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 esse 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 potencial 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 $Area[r][j] \leftarrow Area[r][j] + (\Upsilon[r][w]$ **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, adaptou-se o algoritmo de multiplicação de matrizes descrito em [CUDA Programming Guide 2007], 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 mais rápida do que a memória global.

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 e

o algoritmo sequencial para o cálculo da matriz $Area$ e uma outra versão utilizando programação dinâmica 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 máximo de otimização). 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 e 6GiB de memória global.

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 1 apresenta os resultados obtidos.

Tabela 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 saida.

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)

Os resultados apresentados na Tabela 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 *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 observadores, número de *viewsheds* candidatos e quantidade de células do terreno aumentam.

5. Conclusão

Foi proposta uma heurística de busca local *Swap* eficiente para a solução do problema de posicionamento de observadores. O método proposto combina técnicas de programação dinâmica com programação paralela em CUDA e chega a ser acima 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.

Agradecimentos

Este trabalho foi parcialmente financiado pela FAPEMIG, CNPq e CAPES.

Referências

Ben-Moshe, B., Ben-Shimol, Y., and Y. Ben-Yehezkel, A. Dvir, M. S. (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 *In 7th International Computing and Combinatorics Conference*, pages 121–130, London, UK. Springer-Verlag.

Eidenbenz, S. (2002). Approximation algorithms for terrain guarding. *Inf. Process. Lett.*, 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.

Franklin, W. R. and Vogt, C. (2006). Tradeoffs when multiple observer siting on large terrain cells. In Springer-Verlag, editor, *12th International Symposium on Spatial Data Handling*, pages 845–861.

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

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

CUDA Programming Guide (2007). `http://www.nvidia.com/object/cuda_develop.html` (accessed on Aug 2013).

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

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