Traçamento de raios em GPGPUs

Pedro Geraldo Morelli Rodrigues Alves e Ricardo Biloti, Universidade Estadual de Campinas & INCT-GP, Brasil

Copyright 2011, SBGf - Sociedade Brasileira de Geofísica.

This paper was prepared for presentation at the Twelfth International Congress of the Brazilian Geophysical Society, held in Rio de Janeiro, Brazil, August 15-18, 2011.

Contents of this paper were reviewed by the Technical Committee of the Twelfth International Congress of The Brazilian Geophysical Society and do not necessarily represent any position of the SBGf, its officers or members. Electronic reproduction or storage of any part of this paper for commercial purposes without the written consent of The Brazilian Geophysical Society is prohibited.

Abstract

Neste trabalho falamos sobre a utilização da arquitetura CUDA, criada pela NVIDIA, na simulação de traçamento de raios. Essa arquitetura permite a execução de programas através do processador da placa gráfica, com o intuito de aproveitar a grande capacidade de poder de processamento paralelo que essa possui. Mostramos que, dessa forma o ganho de performance pode chegar a 82% em relação ao algoritmo sequencial.

Introdução

A exploração do meio é um objetivo comum a vários ramos da ciência. Na Sísmica vemos esse objetivo no desejo de estudar as estruturas internas da subsuperfície.

A Teoria dos Raios é uma aproximação de alta frequência para a solução da equação da onda (Miqueles, 2006). Neste sentido, ao invés de computar o campo de onda completo, calculam-se trajetórias preferenciais para a propagação da energia, denominadas raios.

Estas trajetórias obedecem ao sistema de equações diferenciais

$$\frac{\partial x}{\partial t} = v^2 p_1, \tag{1}$$

$$\frac{\partial z}{\partial t} = v^2 p_2, \tag{2}$$

$$\frac{\partial p_1}{\partial t} = \frac{1}{v}\frac{\partial v}{\partial x}, \tag{3}$$

$$\frac{\partial p_2}{\partial t} = \frac{1}{v}\frac{\partial v}{\partial z}, \tag{4}$$

onde $v(x,z)$ é a velocidade de propagação da onda no ponto (x,z) e $p = (p_1, p_2)$ é o vetor vagarosidade, tangente ao raio no ponto (x,z).

Uma maneira de resolver essas equações é através de métodos iterativos como o de Runge-Kutta. Realizar essa tarefa utilizando uma implementação, onde cada raio é calculado de forma não-simultânea, demanda grande quantidade de tempo computacional. Contudo, frequentemente precisa-se realizar o traçamento de milhões de raios, como por exemplo nos algoritmos de migração em profundidade. Desta forma, é importante que seja possível acelerar ao máximo o traçamento de uma enorme massa de raios para que os objetivos desejados sejam alcançados em tempo hábil.

Felizmente, o cálculo de um raio é independente do cálculo de todos os outros raios. Esta característica por si só configura-se como uma grande oportunidade para a paralelização destes cálculos.

Hoje em dia é fácil encontrarmos processadores formados não apenas por um, mas por vários núcleos de processamento chamados *core*. Apesar disso, a capacidade de processamento em paralelo de *CPU*s ainda é muito limitada. As melhores possuem no máximo 8 ou 16 *cores* e um preço bastante elevado. Por outro lado, as unidades de processamento gráfico, *GPU*, evoluíram de maneira diferente. Modelos baratos alcançam facilmente 400 ou 500 *cores* além de uma excelente performance individual de cada *core*. Pensando nisso, a NVIDIA, corporação especializada na fabricação de processadores gráficos, desenvolveu a arquitetura CUDA que permite a criação de programas que serão executados por GPUs, conseguindo dessa forma tirar proveito da enorme capacidade de processamento paralelo que essas GPUs possuem.

Como desvantagem da arquitetura, atualmente CUDA só é suportada por placas gráficas NVIDIA.

Desenvolvimento

Processamento com CUDA

O processamento em paralelo é baseado na divisão do trabalho em *threads* (unidades virtuais de processamento). Na GPU, cada *thread* possui um trecho reservado na memória da GPU e total independência dos demais *threads*. Dessa maneira podemos associar o cálculo da trajetória de cada raio com uma *thread*.

Cada *core* de uma GPU trabalha com grupos de *threads* chamados *warps* (Kirk, 2010). O tamanho de cada *warp* pode variar de acordo com a placa gráfica utilizada, mas geralmente esse número é de 32 *threads* por *warp*. Caso seja necessário criar mais *threads* do que isso, um outro *warp* será criado e associado com outro *core*, caso exista algum livre. Se todos os *cores* estiverem associados a algum *warp* e ainda assim existirem *warps* esperando por processamento, temos uma quebra de paralelismo.

Essa é a limitação do poder de processamento paralelo das GPUs. Uma vez que todos os *cores* estejam operando em seu limite torna-se necessário criar uma fila de espera com os *threads* existentes. Os *threads* que entrarem nessa fila de espera serão processados apenas depois dos *warps* que estiverem em processamento inicialmente (Kirk, 2010).

Um fator que pode gerar ainda mais quebras no paralelismo é que todos os *threads* tem de respeitar pedidos de sincronismo. Caso seja necessário sincronizar os *threads* em uma etapa antes de prosseguir com a execução do algoritmo, o primeiro *warp* que chegar a ela

será enviado ao final da fila e só voltará a ser processado após todos os *warps* que estiverem em sua frente chegarem ao mesmo lugar. Isso ocorre, por exemplo, com a inserção de loops e comandos condicionais no trecho de código em execução na GPU.

Não há controle sobre a ordem de processamento dos *threads* ou sobre a forma com que serão divididos entre *warps*.

Traçador de Raios

Utilizando a Teoria dos Raios e a arquitetura CUDA, desenvolvemos um programa traçador de raios que utiliza o método de Runge-Kutta Fehlberg de quarta ordem (Mathews, 1992) para resolução em paralelo das equações diferenciais (1–4) que descrevem a trajetóriado raio.

Inicialmente carregamos uma malha de velocidades de propagação da onda para um certo modelo. No algoritmo executado está previsto que, caso seja pedido a velocidade de propagação em um certo ponto (x,z) e ela não esteja na malha carregada, a velocidade será obtida através de interpolação da velocidade em pontos próximos que estejam no grid de velocidades. Esse modelo deve ser copiado para a memória da GPU, caso contrário as *threads* não terão acesso a ele.

Cada iteração do método de Runge-Kutta é feita pelos *cores* da GPU através da chamada de um tipo de subrotina computacional chamada *kernel*. Essa subrotina será executada por todos os *threads* criados.

Cada *thread* processa uma única iteração de Runge-Kutta e realiza interpolações da velocidade de propagação em pontos necessários. Quando todos os *threads* tenham concluído a iteração, o controle é devolvido a CPU e ela se incumbe de resgatar os dados processados da memória da placa gráfica para a memória principal.

Existe uma barreira física entre o processamento realizado pela CPU e pela GPU. Elas não conseguem compartilhar memória. Para que a GPU tenha acesso a dados existentes na memória principal é necessário que esses sejam primeiro enviados para a memória da placa gráfica. O mesmo ocorre na volta, para que a CPU tenha acesso a esses dados é necessário que eles sejam copiados de volta da memória da placa gráfica para a memória principal. Essa cópia faz com que exista algum tempo sem processamento perdido onde tanto a GPU quanto a CPU são mantidas em espera.

Esse resultado fica evidente para o traçamento de poucos raios. Nesse caso, os tempos de cópia entre as memórias são relativamente altos, o que torna o algoritmo paralelo mais lento do que o sequencial.

Resultados

Modelo Marmousi

Apresentamos como exemplo o traçamento de raios sobre o modelo Marmousi suavizado (Figura 1). A Figura 2 exibe uma coleção de raios traçados sobre este modelo.

Comparativo de velocidade

Para fins comparativos, o programa traçador de raios pode ser executado de duas formas, uma que utiliza a GPU para o cálculo da trajetória e outra em que apenas a CPU é utilizada. Nessa última utilizamos o método de Runge-Kutta da biblioteca computacional GSL (Gough, 2009).

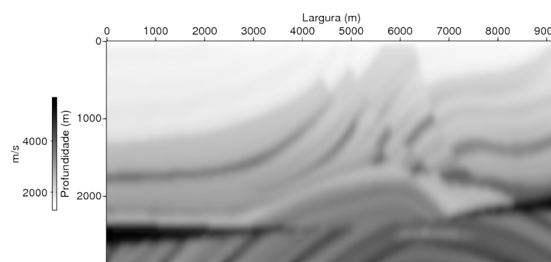

Figure 1: Modelo Marmousi suavizado.

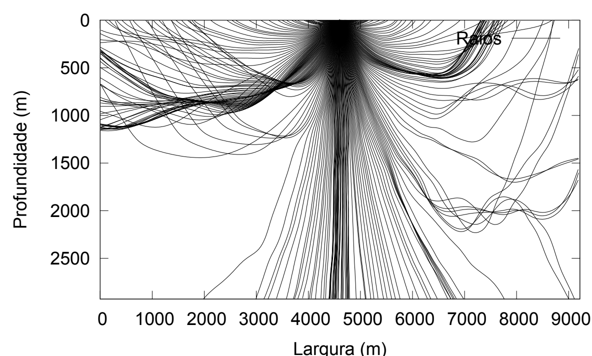

Figure 2: 250 raios traçados a partir da superfície na coordenada horizontal 4600m.

As Figuras 3 e 4 mostram os resultados da execução repetitiva do traçador para quantidades entre 1 e 1.000 raios e 1 e 10.000 raios, respectivamente. Os tempos de processamento para cada quantidade de raios traçada foram ajustados por retas através do método de Quadrados Mínimos (Mathews, 1992) para que fosse mais simples perceber o comportamento qualitativo dos dados. As retas “GPU” e “CPU” representam a relação tempo de processamento versus quantidade de raios traçados.

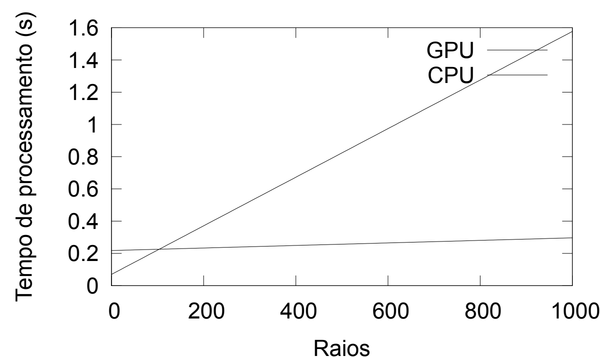

Figure 3: Comparativo do tempo de processamento numa CPU Intel Core 2 Quad Q6600 e uma GPU NVIDIA Geforce GTX 460 para o intervalo entre 1 e 1.000 raios.

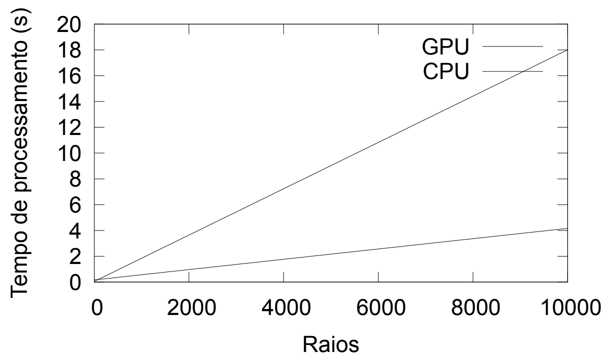

Figure 4: Comparativo do tempo de processamento entre uma CPU Intel Core 2 Quad Q6600 e uma GPU NVIDIA Geforce GTX 460 para o intervalo entre 1 e 10.000 raios.

Como vemos na Figura 3, para poucos raios a simulação é concluída mais lentamente se rodada pela GPU. Isso é uma consequência do tempo de cópia dos dados entre a memória principal e a memória da placa gráfica. Como essa troca deve ocorrer diversas vezes durante a execução da simulação temos que, para poucos raios, a influência desse tempo de cópia se torna mais evidente. Para a execução exclusivamente pela CPU isso não é um problema já que não é necessário realizar a cópia de dados entre memórias.

Por outro lado, vemos ainda mais evidente com a Figura 4 que o ganho da GPU ocorre quando o tamanho do experimento aumenta. Quanto maior o número de raios simulados, mais vantajoso torna-se ao processamento em GPU.

Para a situação observada na Figura 3 temos o processamento de 1 raio em 0.07s enquanto que tivemos 1.000 raios em 1.57s. Isso representa um crescimento de aproximadamente 2210% no tempo de processamento da CPU. Ao mesmo tempo, a GPU calculou 1 raio em 0.22s e 1.000 raios em apenas 0.29s obtendo então um crescimento de 36% no tempo de processamento. Utilizando o algoritmo que tira proveito da GPU pudemos observar uma queda de 82% no tempo de processamento de 1.000 raios em relação a CPU.

Para a situação observada na Figura 4 vemos que a proporção se mantem. A CPU realiza o calculo de 1 raio em 0.05s e de 10.000 raios em 18.01s, representando um aumento de 34.490%. Enquanto isso a GPU realiza o cálculo de 1 raio em 0.15s e de 10.000 raios em 4.02s, ou seja, um aumento de 2.684%. Portanto, utilizando a GPU para traçamento de 10.000 raios tivemos uma queda de 78% no tempo de processamento em relação a CPU.

Conclusões

A utilização da arquitetura CUDA com o objetivo de acelerar o processamento de dados e simulações possui grande potencial. O imenso poder de processamento paralelo que essa arquitetura propõe pode reduzir grandemente o tempo de execução de simulações, como foi visto no caso de traçamento de raios. Apesar de ainda estar amadurecendo, ela já é plenamente funcional e pode acelerar consideravelmente a resolução de problemas geofísicos.

Agradecimentos

Agradecemos ao projeto GêBR e a Petrobras pelo suporte e aos patrocinadores do Consórcio WIT.

Referências

Kirk, D. B., e Hwu, W.W., 2010, Programming massively parallel processors (1st Ed.), Morgan Kaufmann.

Miqueles, E. X., 2006, Modelamento sísmico em meios analíticos, Dissertação(Mestrado em Matemática Aplicada) - Universidade Federal do Paraná.

Mathews, J. H., 1992, Numerical methods for mathematics, science, and engineering(2nd Ed.), Prentice Hall College Div.

Gough, B., 2009, GNU scientific library reference manual (3rd Ed.), Network Theory Ltd.