Análise de desempenho da modelagem de ondas sísmicas acústicas em GPUs com OpenMP

Lucas M. Freire[1], **João B. Fernandes**[1], **Carla Santana**[1], **Samuel Xavier de Souza**[1]

[1]Universidade Federal do Rio Grande do Norte (UFRN) – Natal, RN – Brasil

{lucas.freire,carla.santana,joao.fernandes}.highres@imd.ufrn.br

samuel@dca.ufrn.br

Resumo. *O software Mamute implementa uma solução numérica para a equação da onda, problema com alto custo computacional na sísmica. Este algoritmo rodava originalmente apenas em CPU com multithreading, e sua adaptação em GPU via OpenMP será comentada neste trabalho. Ademais, comparou-se as performances dos compiladores GCC e Clang, com este último sendo sempre superior em GPU.*

1. Introdução

Na sísmica, a modelagem das ondas acústicas consiste na solução da equação da onda via métodos numéricos. Nesse contexto, uma abordagem amplamente utilizada é a aproximação via diferenças finitas, empregando uma discretização do modelo de velocidades em uma grade. Na geofísica, a resolução da grade deve ser suficiente para uso em outros algoritmos como Inversão Completa da Onda (do inglês *Full Waveform Inversion* ou FWI) a fim de atingir a precisão necessária [Yang et al. 2015]. Isto implica em um elevado custo computacional, instigando a aplicação de unidades de processamento gráfico (GPUs), que melhoram a performance de algoritmos do gênero [Yang et al. 2015]. Uma solução pode ser encontrada no *software* Mamute[1], escrito na linguagem C++. O Mamute é *multithread*; contudo, despachos ocorrem em apenas uma *thread* ao usar a GPU. Assim, este trabalho expõe a adaptação em GPU com OpenMP [OpenMP ARB 2015] do Mamute, analisando sua performance e comparando-a com seu desempenho em CPU, considerando diferentes compiladores e precisões numéricas.

2. Modelagem

Para os propósitos da modelagem sísmica acústica, o Mamute implementa uma formulação considerando um meio isotrópico com densidade constante, utilizando o Método das Diferenças Finitas (MDF) de 2ª ordem para derivadas de 2ª ordem temporais e de 8ª ordem para derivadas de 2ª ordem espaciais[1]. Como parâmetros de entrada, tem-se um modelo de velocidades $\mathbf{c}$ e um conjunto de fontes sísmicas nas posições $\{\mathbf{s}_i\}_{i=1}^{n_{src}}$ com assinatura $f(t)$, sendo n_{src} o número de fontes. Ainda, há as coordenadas dos receptores, $\{\mathbf{r}_j\}_{j=1}^{n_{rcv}}$, onde n_{rcv} é a quantidade de receptores. Nelas, séries temporais da magnitude do campo de onda serão registradas, compondo sismogramas.

Cada ponto em $\mathbf{c}$ representa a velocidade de propagação de ondas naquele meio. Assim, considerando condições iniciais nulas e k como o índice de uma amostra temporal, o laço de execução principal atualiza o campo de onda $\mathbf{w}_k$ para $\mathbf{w}_{k+1}$ seguindo o MDF

[1]https://lappsufrn.gitlab.io/seismic/ufrn-fwi/mamute/dev/

proposto, e registra o campo $R_{j,k} = \mathbf{w}_k(\mathbf{r}_j)$ em n_{rcv} pontos, que são as duas principais tarefas do programa. Adicionalmente, consideram-se as condições de contorno para bordas absorventes [Reynolds 1978], um método responsável por minimizar reflexões artificiais nas bordas do modelo, requerindo as matrizes de coeficientes $\mathbf{g}_1$ e $\mathbf{g}_2$.

3. Implementação em GPU

As adaptações do código para execução em GPU podem ser categorizadas em transferências de dados e despacho de *kernels*. Todas as mudanças podem ser verificadas no repositório do projeto[2], com a principal sendo a adição do arquivo `src/ModelingOMPGPU.cpp`. Algumas diretivas do OpenMP utilizadas para alocação e transferência de dados no Mamute foram `#pragma omp target enter data` e `#pragma omp target exit data`. Nas cláusulas `map` de tais diretivas, regras de mapeamento, alocação e transferência foram definidas; assim como o mapeamento de ponteiros. Usando diretivas no formato `enter data map(to: ptr[0:size])`, alocam-se e transferem-se para a GPU $\mathbf{c}$, $\mathbf{g}_1$, $\mathbf{g}_2$, $\{\mathbf{r}_j\}_{j=1}^{n_{rcv}}$, $\mathbf{f}$ que é a discretização de $f(t)$, e o $\mathbf{s}_i$ correspondente à propagação atual. Para cada dado, `ptr` representa seu endereço, enquanto `size` é sua quantidade de elementos. Ainda, espaço é alocado em GPU via a diretiva `enter data map(alloc: ptr[0:size])` para os traços sísmicos de saída $\{\mathbf{R}_j\}_{j=1}^{n_{rcv}}$ e os campos de onda $\mathbf{w}_k$ (acessado via o ponteiro `wave.current`) e $\mathbf{w}_{k+1}$ (acessado via `wave.next`). É importante notar que, para propagações subsequentes, a coordenada da próxima fonte $\mathbf{s}_{i+1}$ é enviada à GPU. No fim do programa, memória em GPU é desalocada com `#pragma omp target exit data map(release: ptr[0:size])`.

O despacho de *kernels* refere-se à execução de código em GPU. Por meio da diretiva `#pragma omp target`, define-se uma região (ou *kernel*) que deve ser executada na placa gráfica. Nela, é possível utilizar outras diretivas de paralelismo, que neste caso foram `teams distribute parallel for`, adicionadas em *kernels* com laços para explorar o modelo de execução paralela da GPU. Em se tratando da atualização do campo de onda, `wave.next` na posição `ind` é computado a partir de `wave.current[ind]` e de outros termos segundo a formulação do MDF[1]. A inserção da fonte em $\mathbf{w}_k$ faz parte da atualização, mas é feita em um *kernel* separado pois só é necessário inserir $\mathbf{f}_k$ em $\mathbf{s}_i$. Por fim, os ponteiros `wave.current` e `wave.next` são trocados em CPU. Na próxima iteração, os novos valores dos ponteiros são remapeados para a GPU, refletindo a mesma troca em seu espaço de endereçamento. Para a leitura do campo, copia-se o valor de `wave.next[ind]` em cada posição $\mathbf{r}_j$ para $\mathbf{R}_{j,k}$. Este estágio também será convertido em um *kernel* e, após a finalização do laço principal, $\{\mathbf{R}_j\}_{j=1}^{n_{rcv}}$ será copiado para o espaço de endereçamento da CPU com uma diretiva `#pragma omp target update from`. Como o mamute suporta a modelagem de várias propagações, mais um *kernel* pode ser necessário para zerar o campo de onda, e o processo descrito anteriormente será repetido.

4. Resultados

Os testes da implementação discutida foram executados em um computador com Processador Intel® Core™ i9-14900. Ele possui 24 núcleos com 32 *threads*. A placa de vídeo

[2] `https://gitlab.com/lappsufrn/seismic/ufrn-fwi/mamute/-/commit/3d26fb638ea3d6e1e98fdd98576f67c8c6436772`

utilizada foi a NVIDIA GeForce RTX™ 4060, com 24 multiprocessadores e 128 núcleos por multiprocessador, totalizando 3072 núcleos. Ademais, o sistema comporta 64 GB de memória RAM, rodando o sistema operacional Ubuntu versão 24.04.2 LTS.

O Mamute foi compilado com CMake versão 3.28.3 e dois compiladores distintos, GCC 13.3.0 e Clang 20.1.6, para comparar suas performances. Outrossim, ele foi testado com precisão simples (32 bits) e dupla (64 bits). Ao compilá-lo para rodar apenas em CPU, ele utiliza todas as *threads* disponíveis da máquina via OpenMP por padrão, e todos os testes em CPU rodaram desta maneira. Das configurações de execução, o modelo de velocidade foi uma matriz cúbica com 101 pontos em cada dimensão, com 10 m de separação entre eles. A velocidade foi constante de 3000 m/s, e o passo de tempo configurado foi de 0,5 ms, com 50 pontos de borda adicionados ao modelo. Apenas uma fonte sísmica foi simulada, um pulso *Ricker* [Ricker 1953] de amplitude 1 Pa e frequência máxima de 20 Hz. Os tempos de execução foram mensurados usando a função `omp_get_wtime`, apenas no laço de modelagem do programa, não contemplando os tempos de transferência de dados da GPU. Para cada configuração de teste, o programa foi rodado 5 vezes, com a mediana dos tempos de execução sendo considerada. Eis o padrão de comando CMake para os testes realizados no Mamute:

```
cmake .. -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=.. \
      -DOMPGPU=(ON|OFF) -DPRECISION=("DOUBLE"|"SINGLE") \
      -DBORDER=DAMPING -DCMAKE_C_COMPILER=(gcc|clang) \
      -DCMAKE_CXX_COMPILER=(g++|clang++) -DOPTIMIZATION=ON -DNATIVE=ON
```

As opções entre parênteses variam com cada configuração de teste, e as opções `OPTIMIZATION` e `NATIVE` habilitam as *flags* `-march=native -ffast-math -fno-cx-limited-range -funroll-all-loops` no caso do GCC. Para o Clang, as *flags* de otimização foram `-march=native -ffast-math -funroll-loops`. Além disso, o *software* NVIDIA Nsight™ Compute [NVIDIA 2025] versão 2025.2.0.0 foi utilizado para perfilar o programa a fim de obter métricas interessantes de performance. Essas perfilagens ocorreram separadamente dos testes mencionados anteriormente. Vide o comando executado para cada perfilagem:

```
ncu --target-processes all --launch-count 10 \
    --section "regex:SpeedOfLight*" bin/mamute modeling \
    --border-type damping --gpu omp test/analytical/modeling.txt
```

A Imagem 1 agrega os dados obtidos. A intensidade aritmética e a performance foram mensuradas no *kernel* da atualização do campo de onda, o mais custoso. A perda de performance do programa compilado com GCC em GPU é notável ao usar precisão dupla, enquanto o Clang foi superior em GPU com ambas as precisões.

Inicialmente, é contra-intuitivo que o programa compilado com GCC possua uma melhor performance em termos de FLOP/s, já que seu tempo de execução foi superior. A outra métrica analisada, intensidade aritmética, auxilia no entendimento desta inconciliabilidade, pois representa uma estimativa da quantidade de operações por byte de dado processado. A diferença é tal porque o GCC precisa realizar mais operações para processar a mesma quantidade de dados, fazendo com que o fluxo de dados que a GPU consome seja menor, mesmo que ele consiga realizar mais FLOPs por segundo.

Outro dado importante é a geometria dos despachos para a GPU. O GCC atribuiu

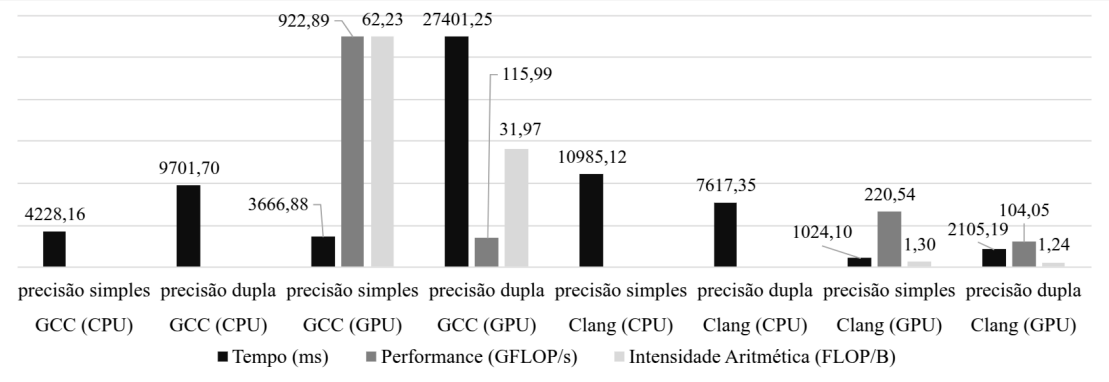

Figura 1. Métricas de desempenho do Mamute

uma *grid* com dimensões $24 \times 1 \times 1$ para precisão dupla e $48 \times 1 \times 1$ para a precisão simples. Em ambos os casos, a dimensão dos blocos é $32 \times 8 \times 1$ *threads*. Já o Clang despacha uma *grid* com dimensões $3200 \times 1 \times 1$ e blocos com dimensões $128 \times 1 \times 1$ com ambas as precisões. A maior quantidade de blocos despachados pelo Clang aumenta as chances de uma boa ocupância ocorrer na GPU. Ou seja, é cabível justificar parte da melhor performance do Clang por ter gerado mais blocos, enquanto o resto da diferença de performance pode ser atribuída a uma melhor geração automática de código. Além disso, a performance superior do GCC em precisão simples em comparação com a dupla também pode ser justificada por mais blocos de *threads*.

5. Conclusão

Foi demonstrado que o compilador Clang entrega performance superior que o GCC em GPU para o Mamute e aplicações similares, potencialmente. É plausível que programas com padrões de acesso à memória similares obtenham resultados congêneres aos apresentados neste trabalho. Portanto, recomenda-se empregar o Clang quando se deseja realizar processamento em placas gráficas utilizando OpenMP. Além disso, é interessante que trabalhos futuros possam investigar mais profundamente o porquê da drástica variação das métricas entre os compiladores.

Referências

[NVIDIA 2025] NVIDIA (2025). NVIDIA NSight™ Compute. `https://developer.nvidia.com/nsight-compute`. Acessado em: 2025-05-16.

[OpenMP ARB 2015] OpenMP ARB (2015). OpenMP application program interface. `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf`. Acessado em: 2025-05-11.

[Reynolds 1978] Reynolds, A. C. (1978). Boundary conditions for the numerical solution of wave propagation problems. *Geophysics*, 43(6):1099–1110.

[Ricker 1953] Ricker, N. (1953). The form and laws of propagation of seismic wavelets. *GEOPHYSICS*, 18(1):10–40.

[Yang et al. 2015] Yang, P., Gao, J., and Wang, B. (2015). A graphics processing unit implementation of time-domain full-waveform inversion. *GEOPHYSICS*, 80(3):F31–F39.