Lucas Morais Freire

# Aplicação de estênceis compactos e reordenamento de dados na modelagem de ondas acústicas

Natal – RN

Dezembro de 2025

Lucas Morais Freire

# Aplicação de estêncils compactos e reordenamento de dados na modelagem de ondas acústicas

Trabalho de Conclusão de Curso de Engenharia de Computação da Universidade Federal do Rio Grande do Norte, apresentado como requisito parcial para a obtenção do grau de Bacharel em Engenharia de Computação

Orientador: Samuel Xavider de Souza

Universidade Federal do Rio Grande do Norte – UFRN

Departamento de Engenharia de Computação e Automação – DCA

Curso de Engenharia de Computação

Natal – RN

Dezembro de 2025

Universidade Federal do Rio Grande do Norte - UFRN
Sistema de Bibliotecas - SISBI
Catalogação de Publicação na Fonte. UFRN - Biblioteca Central Zila Mamede

Freire, Lucas Morais.
Aplicação de estêncils compactos e reordenamento de dados na modelagem de ondas acústicas / Lucas Morais Freire. - 2025.
38 f.: il.

Trabalho de Conclusão de Curso - TCC (graduação) - Universidade Federal do Rio Grande do Norte, Centro de Tecnologia, Curso de Engenharia da Computação e Automação, Natal, RN, 2025.
Orientação: Prof. Dr. Samuel Xavier de Souza.

1. Sísmica - TCC. 2. Modelagem Acústica - TCC. 3. OpenMP - TCC. 4. CUDA - TCC. 5. Cache - TCC. I. Souza, Samuel Xavier de. II. Título.

RN/UF/BCZM CDU 550.834

Elaborado por Fernanda de Medeiros Ferreira Aquino - CRB-15/301

Lucas Morais Freire

# Aplicação de estêncils compactos e reordenamento de dados na modelagem de ondas acústicas

Trabalho de Conclusão de Curso de Engenharia de Computação da Universidade Federal do Rio Grande do Norte, apresentado como requisito parcial para a obtenção do grau de Bacharel em Engenharia de Computação

Orientador: Samuel Xavider de Souza

Trabalho aprovado. Natal – RN, 08 de Dezembro de 2025:

---

**Prof. Samuel Xavier de Souza - Orientador**
UFRN

---

**Prof. Kleiton Andre Schneider - Coorientador**
UFMS

---

**Prof. Tiago Tavares Leite Barros - Convidado**
UFRN

Natal – RN
Dezembro de 2025

*Dedico este trabalho a meus amigos, parentes e, claro, à minha querida Marlucy.*

# AGRADECIMENTOS

Gostaria de agradecer à FUNPEC pela oportunidade de realizar pesquisas na área de processamento de alto desempenho, como também ao professor Kleiton Schneider, que dedicou muito de seu tempo a este trabalho. Também agradeço ao professor Tiago Barros que me inseriu nessa área e ao professor Samuel de Souza pela fé e paciência.

# RESUMO

Este trabalho apresenta motivações, implementações e testes de dois algoritmos de modelagem de ondas acústicas: a Modelagem Acústica Reordenada (MAR) e a Modelagem Acústica Reordenada Compacta (MARC). A modelagem, no geral, é um processo muito importante em diversas áreas da geofísica e do processamento e imageamento sísmicos. Devido a questões de localidade temporal e espacial em memórias cache, estes algoritmos foram desenvolvidos a fim de tentar melhorar o desempenho da modelagem tanto em CPU quanto em GPU com CUDA. Eles foram implementados tendo como base o Mamute, um *software* de processamento e imageamento sísmico, e ambos demonstraram melhoras para tamanhos de problemas maiores quando comparados a outros algoritmos, justificadas pela redução da taxa de *cache-misses*.

**Palavras-chaves**: sísmica; modelagem; OpenMP; CUDA; cache.

# ABSTRACT

This work presents the motivations, implementation details and tests of two acoustic wave modeling algorithms: the Reordered Acoustic Modeling and the Compact Reordered Acoustic Modeling. Seismic wave modeling is an utmost important process in several areas of geophysics and seismic imaging and processing. Due to temporal and spatial cache locality matters, these algorithms were developed to try to improve the performance of wave modeling in both CPUs and GPUs alike. They were implemented using Mamute as a base, which is a seismic imaging and processing software, and both showed performance improvements for larger problem when compared to other algorithms, justified by the reduction of the cache-miss rate.

**Keywords**: seismic; modeling; OpenMP, CUDA; Cache.

# LISTA DE ILUSTRAÇÕES

# LISTA DE TABELAS

# LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| FWI | *Full Waveform Inversion* |
| RTM | *Reverse Time Migration* |
| CPU | *Central Processing Unit* |
| GPU | *Graphics Processing Unit* |
| OpenMP | *Open Multi-Processing* |
| CUDA | *Compute-Unified Device Architecture* |
| MAR | Modelagem Acústica Reordenada |
| MARC | Modelagem Acústica Reordenada Compacta |
| CFL | Courant-Friedrich-Lewis |
| RAM | *Random Access Memory* |
| LRU | *Least Recently Used* |
| FIFO | *First-in First-out* |
| SIMD | *Single Instruction - Multiple Data* |
| WFD1D | *One-dimensional Weights Finite-Difference* |

# SUMÁRIO

# 1 INTRODUÇÃO

A modelagem de ondas acústicas trata de uma solução numérica para uma equação de onda (CARCIONE; HERMAN; KROODE, 2002). No contexto do processamento sísmico, ela é imprescindível para vários outros algoritmos, como Inversão Completa da Onda (do inglês *Full Waveform Inversion* ou FWI), Migração Reversa no Tempo (do inglês *Reverse Time Migration* ou RTM), dentre outros (YANG; GAO; WANG, 2015; BAYSAL; KOSLOFF; SHERWOOD, 1983). Na indústria, Esses algoritmos são comumente utilizados para realizar um Imageamento Sísmico, tarefa cujo objetivo é gerar uma imagem da subsuperfície terrestre, muito utilizados na prospecção de óleo e gás. Em suas versões mais comuns, tanto o FWI quanto o RTM se utilizam de malhas tridimensionais, as quais representam uma discretização da região de interesse, que podem conter milhões de pontos. Portanto, é crucial uma implementação performante e escalável para estas soluções.

Este trabalho, por sua vez, expõe e analisa duas implementações específicas da modelagem tridimensional de ondas acústicas: uma que realiza um reordenamento de dados a fim de tentar melhorar a performance, chamada de Modelagem Acústica Reordenada (MAR); e outra que faz o mesmo, porém utilizando estêncils compactos nas bordas do domínio de resolução, chamada de Modelagem Acústica Reordenada Compacta (MARC), com o intuito de economizar memória quando comparada com a MAR. Ambos foram implementados em C++, utilizando como base o código do Mamute[1], um *software open-source* desenvolvido na Universidade Federal do Rio Grande do Norte; que utiliza OpenMP (OpenMP ARB, 2018) para *multithreading* com memória compartilhada em CPU, e CUDA (NICKOLLS et al., 2008) para execução em GPU. Ambos os algoritmos foram desenvolvidos e testados em CPU com *multithreading* via OpenMP e em GPU via CUDA.

[1] https://gitlab.com/lappsufrn/mamute

# 2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, será apresentada a fundamentação teórica do trabalho, justificando as decisões tomadas e o ferramental matemático disponível.

## 2.1 Modelagem de ondas acústicas

A equação de onda a ser solucionada é explicitada a seguir:

$$\frac{1}{\mathbf{c}^2}\frac{\partial^2 \mathbf{p}}{\partial t^2} - \nabla^2 \mathbf{p} = \mathbf{s} \tag{2.1}$$

Ela supõe um meio isotrópico e perfeitamente elástico, $\mathbf{p}(\mathbf{x}, t)$ representa a magnitude do campo de onda, $\mathbf{c}(\mathbf{x})$ é a velocidade de propagação da onda para cada ponto do espaço, e $\mathbf{s}(\mathbf{x}, t)$ é o termo de fonte.

Ou seja, dado um modelo de velocidades $\mathbf{c}$, um termo de fonte $\mathbf{s}$, condições iniciais e de contorno, pode-se buscar uma solução para esta equação, a fim de prever o comportamento do campo $\mathbf{p}$ ao longo do tempo, para todos os pontos do espaço. Contudo, a depender da função que define o modelo $\mathbf{c}$, soluções analíticas desta equação tornam-se inviáveis, motivando a elaboração de uma solução numérica.

Para realizar tal tarefa, utiliza-se um esquema de diferenças finitas (MOCZO; KRISTEK; GÁLIS, 2014), onde o modelo de velocidades e o campo de onda são discretizados em malhas uniformes, podendo ser abstraídas em matrizes tridimensionais com dimensões $n_\mathrm{x} \times n_\mathrm{y} \times n_\mathrm{z}$. O espaçamento entre pontos será considerado constante por dimensão, respeitando o critério CFL (MOCZO; KRISTEK; GÁLIS, 2014), definindo a relação $\mathbf{c}(\mathbf{x}) = \mathbf{c}(i\Delta x, j\Delta y, k\Delta z) = \mathbf{c}[i, j, k]$, com $i, j, k \in \mathbb{Z}, 0 \leq i < n_\mathrm{x}, 0 \leq j < n_\mathrm{y}, 0 \leq k < n_\mathrm{z}$.

O critério CFL estabelece relações entre o passo de tempo $\Delta t$ e os passos espaciais $\Delta x$,$\Delta y$ e $\Delta z$, a fim de garantir a estabilidade numérica do algoritmo. Existem variações do método, a depender da ordem das discretizações temporais e espaciais, mas as utilizadas no Mamute são baseadas nas estabelecidas por Carcione, Herman e Kroode (2002):

$$\max\{\Delta x, \Delta y, \Delta z\} \leq \frac{c_\mathrm{min}}{2 f_\mathrm{max}} \tag{2.2}$$

$$\Delta t \leq \frac{2 \min\{\Delta x, \Delta y, \Delta z\}}{\pi \sqrt{3} c_\mathrm{max}} \tag{2.3}$$

$f_\mathrm{max}$ é a frequência máxima que se deseja modelar com precisão; $c_\mathrm{max}$ e $c_\mathrm{min}$ são as velocidades máxima e mínima em $\mathbf{c}$, respectivamente. Com base nisso, discretiza-se o tempo

escolhendo um $\Delta t$ apropriado segundo este critério, e define-se $t = n\Delta t$, com $0 \leq n < N$. Por fim, o laplaciano será calculado via uma aproximação centrada de oitava ordem para as derivadas segundas espaciais, e a segunda derivada temporal do campo de onda será computada a partir de uma aproximação centrada de segunda ordem (FORNBERG, 1988); seguindo o padrão já estabelecido no Mamute. Logo, a seguinte equação apresenta a versão discretizada da Equação 2.1:

$$\frac{1}{\mathbf{c}^2}\frac{\mathbf{p}^{n-1} - 2\mathbf{p}^n + \mathbf{p}^{n+1}}{\Delta t^2} - \nabla^2_{8^a}\mathbf{p}^n = \mathbf{s}^n \tag{2.4}$$

A partir disso, isola-se o termo $\mathbf{p}^{n+1}$, para obter uma relação recursiva:

$$\mathbf{p}^{n+1} = 2\mathbf{p}^n - \mathbf{p}^{n-1} + \mathbf{c}^2\Delta t^2\nabla^2_{8^a}\mathbf{p}^n + \mathbf{c}^2\Delta t^2\mathbf{s}^n \tag{2.5}$$

Entretanto, diversas considerações práticas devem ser feitas sobre esta discretização, a fim de torná-la útil. Por exemplo, a aproximação do laplaciano é calculada de acordo com a relação:

$$\nabla^2_{8^a}\mathbf{p}^n[i,j,k] = \sum_{d\,=\,-4}^{4} a_d \left( \frac{1}{\Delta x^2}\mathbf{p}^n[i+d,j,k] + \frac{1}{\Delta y^2}\mathbf{p}^n[i,j+d,k] + \frac{1}{\Delta z^2}\mathbf{p}^n[i,j,k+d] \right) \tag{2.6}$$

Isso representa a soma das três derivadas segundas espaciais, centradas no ponto $[i,j,k]$, e os coeficientes de diferenças finitas são representados por $a_d$. É explícito que o laplaciano para uma posição $[i,j,k]$ depende do valor de $\mathbf{p}$ nas posições $[i \pm d, j \pm d, k \pm d]$, com $d \in \mathbb{Z}, 0 \leq d \leq 4$, um conjunto de pontos que possui um formato comumente chamado de estêncil. Por conseguinte, em um cenário computacional que usa este esquema, não se pode computar o laplaciano para posições próximas das bordas do domínio de simulação, porque demandaria o acesso de posições não contempladas pelas dimensões das matrizes, tornando impossível atualizar as camadas mais externas do campo de onda via a Equação 2.5 em conjunto com a Equação 2.6.

Outra questão é a imposição de condições de contorno de Dirichlet, que causam reflexões nas bordas, característica indesejada quando o objetivo é simular um meio que aparenta infinito em todas as direções. Existem algumas soluções possíveis para esse problema, mas, para atingir o objetivo maior de apresentar os algoritmos de reordenamento, Bordas Absorventes (do inglês *Damping Borders*) serão suficientes, e encontram-se implementadas no Mamute. Este método envolve inserir coeficientes de amortecimento $\mathbf{g}(\mathbf{x})$, que atenunam a amplitude de $\mathbf{p}^{n-1}$ e $\mathbf{p}^{n+1}$ segundo o método descrito por Cerjan et al. (1985).

$$\mathbf{p}^{n+1} = \mathbf{g}\left(2\mathbf{p}^{n} - \mathbf{g}\mathbf{p}^{n-1} + \mathbf{c}^2\Delta t^2\nabla^2_{8^{\mathrm{a}}}\mathbf{p}^{n} + \mathbf{c}^2\Delta t^2\mathbf{s}^{n}\right) \tag{2.7}$$

Para se beneficiar do efeito das bordas absorventes, O domínio de simulação é expandido em todos os lados em $n_{\mathrm{b}}$ pontos. Dessa forma, o domínio expandido fica com dimensões totais definidas por:

$$e_{\mathrm{nx}} = n_{\mathrm{x}} + 2n_{\mathrm{b}} \tag{2.8}$$

$$e_{\mathrm{ny}} = n_{\mathrm{y}} + 2n_{\mathrm{b}} \tag{2.9}$$

$$e_{\mathrm{nz}} = n_{\mathrm{z}} + 2n_{\mathrm{b}} \tag{2.10}$$

Para a aplicação do Mamute, entende-se que o termo $\mathbf{s}$ só é não-nulo em um ponto do espaço: nas coordenadas da fonte. Consequentemente, o algoritmo final dedica um passo isolado, comumente chamado de inserção da fonte, para que o termo da fonte seja somado no ponto $\mathbf{x}_{\mathrm{s}}$ apenas. Ainda, considera-se que $\mathbf{s}$ é uma grandeza de densidade energética, e não de pura magnitude. Portanto, ao adicionar uma fonte com assinatura $f(t)$, tal que $f^n = f(n\Delta t)$, a multiplicamos por $1/(\Delta x\Delta y\Delta z)$ para só então inserí-la (MOCZO; KRISTEK; GÁLIS, 2014). Assim, obtém-se a versão final da equação de atualização, com a última parcela representando o valor adicionado no passo de inserção de fonte:

$$\mathbf{p}^{n+1} = \mathbf{g}\left(2\mathbf{p}^{n} - \mathbf{g}\mathbf{p}^{n-1} + \mathbf{c}^2\Delta t^2\nabla^2_{8^{\mathrm{a}}}\mathbf{p}^{n}\right) + \mathbf{g}\mathbf{c}^2\Delta t^2\frac{\delta(\mathbf{x}-\mathbf{x}_{\mathrm{s}})f^n}{\Delta x\Delta y\Delta z} \tag{2.11}$$

Com estas considerações, há ferramentas suficientes para implementar este algoritmo. Porém, ainda precisa ser discutida a saída do programa resultante, que será um conjunto de traços sísmicos registrados em posições arbitrárias do domínio, chamados de receptores. Portanto, define-se a matriz $\mathbf{r}$, onde $\mathbf{r}[m,n]$ é o valor de $\mathbf{p}^n$ registrado no $m$-ésimo receptor, considerando $M$ receptores. As coordenadas $x$, $y$ e $z$ destes receptores se encontram nos vetores $\mathbf{r}_{\mathrm{x}}$,$\mathbf{r}_{\mathrm{y}}$ e $\mathbf{r}_{\mathrm{z}}$, respectivamente. Assim, define-se o Algoritmo 1.

Quanto a detalhes mais específicos sobre o Mamute, no presente momento, consideram-se condições iniciais nulas para o campo de onda, e apenas duas matrizes são utilizadas para representá-lo: uma chamada de `current` e outra chamada de `next`. `current` sempre faz o papel de $\mathbf{p}^n$ enquanto `next` faz o papel de ambos $\mathbf{p}^{n+1}$ e $\mathbf{p}^{n-1}$. Isso é possível porque, antes do passo da linha 7 do Algoritmo 1, `next` contém os valores de $\mathbf{p}^{n-1}$, e durante o processo executado na linha 7 seus valores são substituídos pelos correspondentes a $\mathbf{p}^{n+1}$. No fim de cada iteração do laço mais externo, os ponteiros de `current` e `next` são trocados, para prepará-los para a próxima iteração.

**Algoritmo 1:** Modelagem Acústica

**Entrada:** $\mathbf{C}, f, \mathbf{x}_\mathrm{s}, \mathbf{r}_\mathrm{x}, \mathbf{r}_\mathrm{y}, \mathbf{r}_\mathrm{s}$
**Saída:** **r**

$\mathbf{g} \leftarrow$ computarCoeficientesDamping();
$\mathbf{p}^{-1} \leftarrow \mathbf{0}$;
$\mathbf{p}^{0} \leftarrow \mathbf{0}$;
$n \leftarrow 0$;
**enquanto** $n \leq N$ **faça**
    `// Via Equação 2.6:`
    $\nabla^2\mathbf{p}^n \leftarrow$ calcularLaplaciano($\mathbf{p}^n$);
    $\mathbf{p}^{n+1} \leftarrow \mathbf{g}\left(2\mathbf{p}^n - \mathbf{g}\mathbf{p}^{n-1} + \mathbf{c}^2\Delta t^2\nabla^2\mathbf{p}^n\right)$;
    `// Inserção da fonte:`
    $\mathbf{p}^{n+1}(\mathbf{x}_\mathrm{s}) \leftarrow \mathbf{p}^{n+1}(\mathbf{x}_\mathrm{s}) + \mathbf{g}\mathbf{c}^2\Delta t^2 f^n/(\Delta x\Delta y\Delta z)$;
    `// Leitura dos receptores:`
    $m \leftarrow 0$;
    **enquanto** $m \leq M$ **faça**
        $\mathbf{r}[m,n] \leftarrow \mathbf{p}^n(\mathbf{r}_\mathrm{x}[m], \mathbf{r}_\mathrm{y}[m], \mathbf{r}_\mathrm{z}[m])$;
        $m \leftarrow m + 1$;
    **fim**
    $n \leftarrow n + 1$;
**fim**
**retorna** ***r;***

## 2.2 O problema da memória cache

A memória cache é um dispositivo de armazenamento volátil de rápido acesso (HENNESSY; PATTERSON, 2017). A grande maioria dos dados que um processador recebe é carregado na memória cache primeiro. Por conseguinte, o mecanismo da cache influencia diretamente na performance de um programa, ainda mais considerando que ela se tornou onipresente em arquiteturas de computadores atuais. Usualmente, a cache possui três níveis: L1, L2 e L3, com o L3 sendo o maior deles, compartilhado por todos os núcleos do processador. Os níveis L1 e L2, por sua vez, são menores e específicos de um núcleo.

O mecanismo da cache pode ser abstraído em dois conceitos: a localidade espacial e a localidade temporal. A localidade espacial refere-se ao fato de que, ao tentar carregar dados advindos da RAM, a memória cache também carrega dados vizinhos ao que foi originalmente requisitado, fazendo com que o acesso às posições próximas seja mais rápido. A localidade temporal, por sua vez, envolve o mecanismo da política de substituição da cache. Muitas políticas diferentes existem, como LRU e FIFO (HENNESSY; PATTERSON, 2017), mas elas geralmente possuem um efeito em comum: a tendência de reter dados acessados mais recentemente. Isso beneficia algoritmos que acessem um mesmo conjunto de blocos com alta frequência.

No contexto da modelagem de ondas acústicas via diferenças finitas, a cache se prova um gargalo, como será constatado na seção 4.4. O problema se encontra na maneira como dados são acessados durante o cálculo do laplaciano, porque o padrão de acesso é bastante hostil às localidades temporal e espacial dependendo de como matrizes são estruturadas na memória.

O arranjo mais comum para uma matriz em C ou C++ é o chamado de *row-major*, onde elementos em uma linha são contíguos na memória. Por exemplo, considere a matriz `float A[20][10]`, com 20 linhas e 10 colunas. Para acessar uma elemento na posição $[i, j]$ desta matriz, a expressão `A[i][j]` é idêntica à expressão `*(A[0] + 10*i + j)`, pois há um passo (ou *stride*) de 10 elementos entre cada linha. Para o laplaciano, acessos na direção com menor *stride*, que no caso do exemplo é a dimensão do índice $j$, são amigáveis para com a cache; mas qualquer outra direção irá acessar um elemento possivelmente muito mais longe na memória. Isso é mais agravante em *softwares* como o Mamute, onde modelos podem ter milhões de pontos. À medida em que o ponto no qual o laplaciano é calculado varia, dados em linhas paralelas próximas são carregados, mas só voltam a ser acessados quando a próxima linha ou plano da matriz for percorrido, como exemplifica a Figura 1.

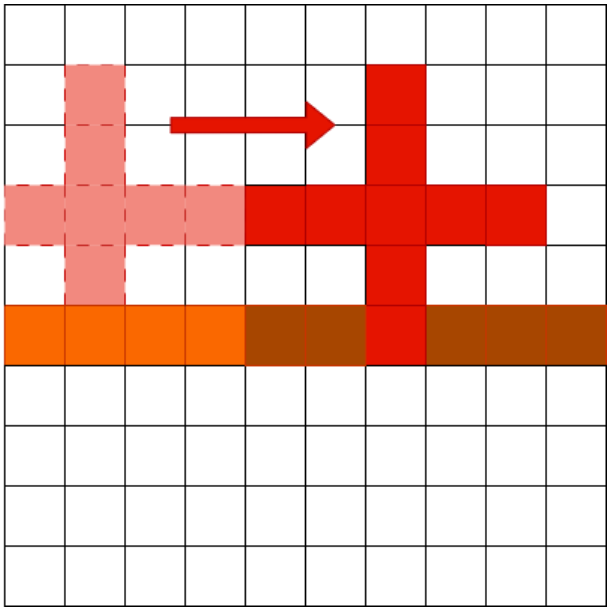

Figura 1 – Os blocos laranja e marrom são mapeados para linhas de cache distintas. A cruz vermelha representa o estêncil percorrendo a matriz, de tal forma que o bloco laranja só será acessado novamente quando a próxima linha da matriz for percorrida.

Em matrizes tridimensionais este problema é ainda mais agravante. Pensando nisso, soluções diferentes foram concebidas, sendo o ladrilhamento uma das mais conhecidas (WOLF; LAM, 1991). O ladrilhamento consiste em modificar a ordem em que um laço acessa elementos da matriz, a fim de melhorar a localidade temporal. Com este método, a

matriz é subdividida em blocos, e cada bloco é internamente percorrido na mesma ordem de um laço usual. As Figuras 2 e 3 exemplificam este comportamento.

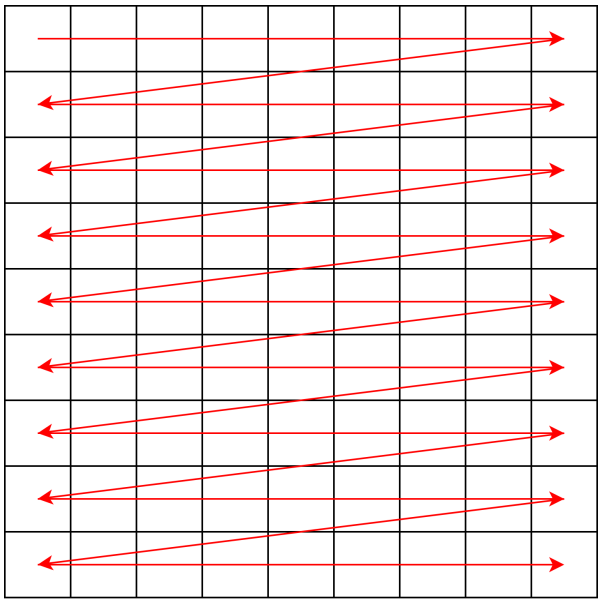

Figura 2 – Ordem de acesso padrão.

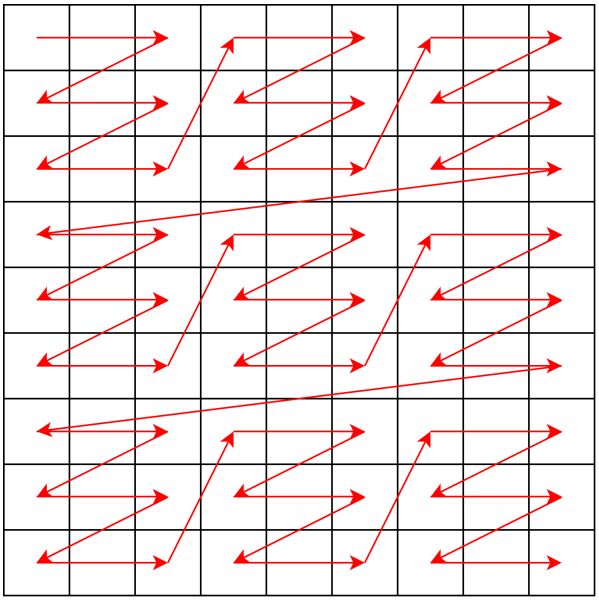

Figura 3 – Ordem de acesso do ladrilhamento.

Esta técnica é efetiva, mas seria interessante melhorar a localidade espacial juntamente da localidade temporal, problema central deste trabalho, explorado mais à fundo na seção seguinte.

## 2.3 Reordenamento

Na Figura 3, as setas indicam a ordem de acesso do ladrilhamento, enquanto a Figura 2 não só indica a ordem de acesso comum, como também coincide com a maneira como elementos estão dispostos em memória, com as setas representando endereços em ordem crescente. Para a solução proposta neste trabalho, o inverso deve acontecer, com a Figura 2 indicando a ordem de acesso, e a Figura 3 indicando a maneira como os dados estão dispostos em memória. Dessa forma, a ordem de acesso permanece linear, mas os dados em si estão ordenados de forma a manter pontos próximos tanto espacialmente (segundo a abstração de uma matriz) quanto em memória. Bibliotecas como BrickLib (ZHAO et al., 2018) fazem processos deste tipo, mas usam blocos que não são contíguos em memória, e compensa isso utilizando *halos* em volta de cada bloco, que devem ser trocados com seus vizinhos entre iterações do laço principal de modelagem. Esta abordagem favorece o uso de instruções SIMD, mas quebra a contiguidade entre blocos.

Como ponto de partida, define-se o tamanho da lateral de um bloco como $b_{sz}$, e assumindo o mesmo estêncil de ordem 8, com 9 pontos, define-se $b_{sz} \geq 9$. Esta escolha facilita

a implementação, mas também se demonstra razoável se o contrário fosse considerado. Caso $b_{\text{sz}} < 9$, a todo momento teria-se um estêncil buscando dados em blocos vizinhos, o que frustra o propósito do método em primeiro lugar. Com isso definido, a quantidade de blocos para cada dimensão $(n_{\text{bx}}, n_{\text{by}}, n_{\text{bz}})$ necessários para cobrir todo o modelo com bordas inclusas podem ser calculadas da seguinte forma:

$$n_{\text{bx}} = \left\lceil \frac{e_{\text{nx}}}{b_{\text{sz}}} \right\rceil \tag{2.12}$$

$$n_{\text{by}} = \left\lceil \frac{e_{\text{ny}}}{b_{\text{sz}}} \right\rceil \tag{2.13}$$

$$n_{\text{bz}} = \left\lceil \frac{e_{\text{nz}}}{b_{\text{sz}}} \right\rceil \tag{2.14}$$

Então, analisa-se o *stride* entre elementos. Caso um ponto e seu vizinho estejam no mesmo bloco, a distância entre ambos seria 1 no eixo $z$. Enquanto isso, nos eixos $x$ e $y$, o *stride* seria de $b_{\text{sz}}^2$ e $b_{\text{sz}}$ respectivamente, já que o bloco é contíguo. Este resultado é sumarizado nas seguintes equações:

$$s_{\text{px}} = b_{\text{sz}}^2 \tag{2.15}$$

$$s_{\text{py}} = b_{\text{sz}} \tag{2.16}$$

$$s_{\text{pz}} = 1 \tag{2.17}$$

$(s_{\text{px}}, s_{\text{py}}, s_{\text{pz}})$ representam os *strides* em cada dimensão, entre pontos em um mesmo bloco. Para compreender como funciona o *stride* entre pontos em blocos diferentes, considere novamente a Figura 3. Definindo $b_{\text{vol}}$ como a quantidade de elementos em um bloco, e $I$ como o passo a partir do ponteiro base da matriz, representando uma coordenada $[i, j, k]$ qualquer, é notável que qualquer $I$ somado com $b_{\text{vol}}$ resulta em uma posição $I_2$ no bloco seguinte, com a mesma posição relativa ao início do respectivo bloco, efetivamente incrementando a coordenada de menor *stride* por $b_{\text{sz}}$. Supondo que o eixo com menor *stride* é o $z$, e aquele com maior *stride* é o $x$, para atingir o mesmo efeito de incrementar sua coordenada em $b_{\text{sz}}$, os saltos devem ser de $n_{\text{by}} \cdot n_{\text{bz}} \cdot b_{\text{vol}}$ e $n_{\text{bz}} \cdot b_{\text{vol}}$ elementos respectivamente para $x$ em $y$.

$$s_{\text{bx}} = n_{\text{by}} n_{\text{bz}} b_{\text{vol}} \tag{2.18}$$

$$s_{\text{by}} = n_{\text{bz}} b_{\text{vol}} \tag{2.19}$$

$$s_{\text{bz}} = b_{\text{vol}} \tag{2.20}$$

Assim, caso um ponto $[i, j, k]$ se encontre próximo o suficiente da borda de um bloco arbitrário A, e deseje-se acessar a posição $[i + d, j, k]$ que está no bloco vizinho B, encontra-

se primeiro o índice linear $I$ a partir de $[i, j, k]$ (processo que será discutido posteriormente) e calcula-se $I + d \cdot s_{\text{bx}} = I_{\text{b}}$, onde $I_{\text{b}}$ é o índice linear do ponto $[i + b_{\text{sz}}, j, k]$, como discutido anteriormente. A partir deste novo índice, translada-se o ponto mais uma vez, só que por $d - b_{\text{sz}}$ ao longo do eixo $x$ utilizando o *stride* $s_{\text{px}}$. Esta translação é garantida de resultar em um ponto no mesmo bloco B, pois $d \leq b_{\text{sz}}$ e $[i + b_{\text{sz}} + d - b_{\text{sz}}] = [i + d, j, k]$. A mesma lógica se aplica para outras dimensões. Logo, dado um ponto $[i, j, k]$ que corresponde a um índice linear $I$, as equações seguintes indicam como computar um *offset* correspondente para uma translação por $d$ elementos em alguma dimensão, com $d \leq b_{\text{sz}}$:

$$[i + d, j, k] : \begin{cases} \text{se } i \bmod b_{\text{sz}} + d \geq b_{\text{sz}}, I + s_{\text{bx}} - (b_{\text{sz}} - d)s_{\text{px}} \\ \text{se } i \bmod b_{\text{sz}} + d < b_{\text{sz}}, I + ds_{\text{px}} \end{cases} \tag{2.21}$$

$$[i - d, j, k] : \begin{cases} \text{se } i \bmod b_{\text{sz}} \geq d, I - ds_{\text{px}} \\ \text{se } i \bmod b_{\text{sz}} < d, I - s_{\text{bx}} + (b_{\text{sz}} - d)s_{\text{px}} \end{cases} \tag{2.22}$$

$$[i, j + d, k] : \begin{cases} \text{se } j \bmod b_{\text{sz}} + d \geq b_{\text{sz}}, I + s_{\text{by}} - (b_{\text{sz}} - d)s_{\text{py}} \\ \text{se } j \bmod b_{\text{sz}} + d < b_{\text{sz}}, I + ds_{\text{py}} \end{cases} \tag{2.23}$$

$$[i, j - d, k] : \begin{cases} \text{se } j \bmod b_{\text{sz}} \geq d, I - ds_{\text{py}} \\ \text{se } j \bmod b_{\text{sz}} < d, I - s_{\text{by}} + (b_{\text{sz}} - d)s_{\text{py}} \end{cases} \tag{2.24}$$

$$[i, j, k + k] : \begin{cases} \text{se } k \bmod b_{\text{sz}} + d \geq b_{\text{sz}}, I + s_{\text{bz}} - (b_{\text{sz}} - d)s_{\text{pz}} \\ \text{se } k \bmod b_{\text{sz}} + d < b_{\text{sz}}, I + ds_{\text{pz}} \end{cases} \tag{2.25}$$

$$[i, j, k - d] : \begin{cases} \text{se } k \bmod b_{\text{sz}} \geq d, I - ds_{\text{pz}} \\ \text{se } k \bmod b_{\text{sz}} < d, I - s_{\text{bz}} + (b_{\text{sz}} - d)s_{\text{pz}} \end{cases} \tag{2.26}$$

Assim, acessos a pontos vizinhos se torna um desafio nessa formulação, pois condicionais seriam complexas e causariam um impacto visível de performance dentro de um laço de cálculo do laplaciano. Pensando nisso, foi decidido pré-calcular os *offsets* seguindo as Equações 2.21 a 2.26. Como o layout dentro de cada bloco é idêntico, só é necessário computar *offsets* para $0 \leq p < b_{\text{sz}}, 1 \leq d \leq 4$ uma vez, tanto para a versão de incremento, como em 2.21, quanto para a versão de decremento, como em 2.22.

Para isso, é preciso conseguir extrair um índice linear $I$ a partir de coordenadas de um bloco $(b_i, b_j, b_k)$, e coordenadas $(p_i, p_j, p_k)$ de um ponto em relação à base do bloco. Com isso, temos a Equação 2.27.

$$I = s_{\text{bx}}b_i + s_{\text{by}}b_j + s_{\text{bz}}b_k + s_{\text{px}}p_i + s_{\text{py}}p_j + p_k \tag{2.27}$$

Com estas ferramentas em mãos, os laços de execução podem ser escritos. Primeiro, itera-se sobre os blocos, e em cada bloco, itera-se sobre os pontos. Utilizando a equação

2.27, computa-se o índice linear, que pode ser incrementado ou decrementado com valores pré-calculados seguindo as Equações 2.21 até 2.26 para acessar elementos em todas as direções, a fim de computar o laplaciano.

Já que não é possível restringir os limites do laço com esse esquema para ignorar pontos próximos às bordas, é preciso lidar com essa questão de forma diferente. Seria possível apenas computar as coordenadas $[i, j, k]$ a partir dos índices dos blocos e dos pontos, e construir uma condição que rejeita pontos próximos às bordas. Porém, a fim de tentar empurrar o máximo possível de processamento nesses laços, a estratégia escolhida será a de adicionar uma camada extra de blocos em todas as direções. Assim, pode-se iterar pelos blocos de índice $1...n_{\text{bx}} - 2$, por exemplo, e usar pontos dos blocos 0 ou $n_{\text{bx}} - 1$ quando estiver próximo de uma borda. Isso também faz com que o algoritmo possivelmente percorra mais pontos além da borda definida pelo usuário. Portanto, todos os blocos devem ser devidamente inicializados, assim como os coeficientes de bordas absorventes para todos os pontos em todos os blocos. Para descobrir a quantidade de blocos necessária para cobrir o domínio dessa forma, ignoramos os 4 pontos em cada lado para cada dimensão, já que não fazem parte do domínio iterável. Então, para a Modelagem Acústica Reordenada, utilizam-se as quantidades de blocos definidas pelas equações a seguir:

$$n_{\text{bx}} = \left\lceil \frac{e_{\text{nx}} - 8}{b_{\text{sz}}} \right\rceil + 2 \tag{2.28}$$

$$n_{\text{by}} = \left\lceil \frac{e_{\text{ny}} - 8}{b_{\text{sz}}} \right\rceil + 2 \tag{2.29}$$

$$n_{\text{bz}} = \left\lceil \frac{e_{\text{nz}} - 8}{b_{\text{sz}}} \right\rceil + 2 \tag{2.30}$$

Uma solução alternativa que evita o uso dessa camada extra é utilizar coeficientes de diferenças finitas não centrados nas bordas, os chamados de estêncils compactos.

## 2.4 Formulação compacta

Fornberg (1988), Carrillo, Schneider e Parés (2025) descrevem métodos para se conseguir coeficientes de diferenças finitas não-centrados. Em especial, o WFD1D é um algoritmo recursivo capaz de gerar coeficientes para uma quantidade arbitrária de pontos. Continua-se utilizando uma aproximação com 9 pontos, mas foi verificado que computar derivadas para um nó fora do centro do conjunto de pontos reduz a ordem de precisão para 7, em contraste com a ordem 8 quando o esquema é central. Assim, o WDF1D foi empregado para gerar os coeficientes da Tabela 1.

O uso do esquema compacto faz o algoritmo tratar as bordas de maneira especial, obrigando o uso de condicionais para controlar se será usado diferenças finitas centrais ou

| $i$ | Coeficientes | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | $\frac{29531}{5040}$ | $\frac{-962}{35}$ | $\frac{621}{10}$ | $\frac{-4006}{45}$ | $\frac{691}{8}$ | $\frac{-282}{5}$ | $\frac{2143}{90}$ | $\frac{-206}{35}$ | $\frac{363}{560}$ |
| 1 | $\frac{363}{560}$ | $\frac{8}{315}$ | $\frac{-83}{20}$ | $\frac{153}{20}$ | $\frac{-529}{72}$ | $\frac{47}{10}$ | $\frac{-39}{20}$ | $\frac{599}{1260}$ | $\frac{-29}{560}$ |
| 2 | $\frac{-29}{560}$ | $\frac{39}{35}$ | $\frac{-331}{180}$ | $\frac{1}{5}$ | $\frac{9}{8}$ | $\frac{-37}{45}$ | $\frac{7}{20}$ | $\frac{-3}{35}$ | $\frac{47}{5040}$ |
| 3 | $\frac{47}{5040}$ | $\frac{-19}{140}$ | $\frac{29}{20}$ | $\frac{-118}{45}$ | $\frac{11}{8}$ | $\frac{-1}{20}$ | $\frac{-7}{180}$ | $\frac{1}{70}$ | $\frac{-1}{560}$ |
| 4 | $\frac{-1}{560}$ | $\frac{8}{315}$ | $\frac{-1}{5}$ | $\frac{8}{5}$ | $\frac{-205}{72}$ | $\frac{8}{5}$ | $\frac{-1}{5}$ | $\frac{8}{315}$ | $\frac{-1}{560}$ |
| 5 | $\frac{-1}{560}$ | $\frac{1}{70}$ | $\frac{-7}{180}$ | $\frac{-1}{20}$ | $\frac{11}{8}$ | $\frac{-118}{45}$ | $\frac{29}{20}$ | $\frac{-19}{140}$ | $\frac{47}{5040}$ |
| 6 | $\frac{47}{5040}$ | $\frac{-3}{35}$ | $\frac{7}{20}$ | $\frac{-37}{45}$ | $\frac{9}{8}$ | $\frac{1}{5}$ | $\frac{-331}{180}$ | $\frac{39}{35}$ | $\frac{-29}{560}$ |
| 7 | $\frac{-29}{560}$ | $\frac{599}{1260}$ | $\frac{-39}{20}$ | $\frac{47}{10}$ | $\frac{-529}{72}$ | $\frac{153}{20}$ | $\frac{-83}{20}$ | $\frac{8}{315}$ | $\frac{363}{560}$ |
| 8 | $\frac{363}{560}$ | $\frac{-206}{35}$ | $\frac{2143}{90}$ | $\frac{-282}{5}$ | $\frac{691}{8}$ | $\frac{-4006}{45}$ | $\frac{621}{10}$ | $\frac{-962}{35}$ | $\frac{29531}{5040}$ |

Tabela 1 – Pesos para uma derivada de segunda ordem em diferenças finitas usando 9 pontos, gerados pelo algoritmo WFD1D. À esquerda, o índice $i$ indica qual é o índice do peso que corresponde à posição do ponto que terá sua segunda derivada estimada.

diferenças finitas descentralizadas. Também é razoável utilizar as mesmas quantidades de blocos estipuladas pelas Equações 2.12 a 2.14. Isso faz com que a Modelagem Acústica Reordenada Compacta economize mais memória quando comparada à versão não-compacta, em média.

# 3 IMPLEMENTAÇÃO

Neste capítulo, será detalhada a implementação dos algoritmos no Mamute, a fim de esclarecer detalhes práticos. Códigos mostrados neste capítulo serão baseados em suas versões reais do Mamute, e algumas variáveis terão nomes diferentes a fim de facilitar o entendimento dos códigos.

## 3.1 Tratamento do modelo de velocidades

No Mamute, quando o modelo de velocidades é carregado em memória, ele naturalmente possui um *layout* próprio para a iteração utilizando o método comum, com destaque no desconto dado em cada borda por causa do estêncil:

```
for (uint i = 4; i < enx - 4; ++i) {
  for (uint j = 4; i < eny - 4; ++j) {
    for (uint k = 4; i < enz - 4; ++k) {
      // Processar
    }
  }
}
```

Mas, para utilizar a Modelagem Acústica Reordenada, precisa-se reordenar o modelo de velocidades. O processo de reordenamento também extrapola os valores nas bordas, copiando valores da parte mais externa em todas as direções para preencher o volume completo do modelo expandido. Este algoritmo que também é feito na Modelagem comum, porém em uma execução separada da expansão em si. Na MAR, o laço de reordenamento é dado por:

```
#pragma omp parallel for collapse(6)
for (size_t bi = 0; bi < nbx; bi++) {
  for (size_t bj = 0; bj < nby; bj++) {
    for (size_t bk = 0; bk < nbz; bk++) {
      for (size_t pi = 0; pi < bsz; pi++) {
        for (size_t pj = 0; pj < bsz; pj++) {
          for (size_t pk = 0; pk < bsz; pk++) {

            // Computar indices globais
            size_t i = bi*bsz + pi;
            size_t j = bj*bsz + pj;
            size_t k = bk*bsz + pk;

            if (i < rstartx) {
```

```
15                i = 0;       // Expandir para -x
16              } else if (i >= rendx) {
17                i = nx - 1; // Expandir para +x
18              } else {
19                i = i - rstartx;
20              }
21
22              if (j < rstarty) {
23                j = 0;       // Expandir para -y
24              } else if (j >= rendy) {
25                j = ny - 1; // Expandir para +y
26              } else {
27                j = j - rstarty;
28              }
29
30              if (k < rstartz) {
31                k = 0;       // Expandir para -z
32              } else if (k >= rendz) {
33                k = nz - 1; // Expandir para +z
34              } else {
35                k = k - rstartz;
36              }
37
38              size_t r_pind = pi*psx + pj*psy + pk;
39              size_t r_ind = bi*bsz + bj*bsy + bk*bsz + r_pind;
40              size_t ind   = (i*ny + j)*nz + k;
41              modelo_reordenado[r_ind] = modelo_original[ind];
42            }
43          }
44        }
45      }
46    }
47  }
```

Observe que as iterações já estão no formato de 6 laços que o modelo reordenado usará. Isto foi feito para reduzir o possível *false sharing* que ocorreria caso o laço seguisse a ordem natural do modelo original. Da forma apresentada, cada *thread* escreve sequencialmente na memória, sem dar saltos , pois o escalonamento padrão do OpenMP é *static*. Claro, o acesso à matriz original será mais caótico, mas considera-se a escrita uma operação mais agressiva por causa da possibilidade de *false sharing*.

No início de cada iteração, as coordenadas globais em relação à matriz reordenada são computadas, com a intenção de mapeá-las para coordenadas equivalentes no modelo original. Como a matriz reordenada é maior que a original, *offsets* são necessários para fazer o mapeamento entre coordenadas das duas matrizes. Estes *offsets* são `rstartx`, `rstarty`

e `rstartz`. Como a matriz original é mapeada para a reordenada de forma centralizada, estes valores podem ser obtidos da seguinte maneira:

$$r_{\text{startx}} = \left\lfloor \frac{n_{\text{bx}} b_{\text{sz}} - n_{\text{x}}}{2} \right\rfloor \tag{3.1}$$

$$r_{\text{starty}} = \left\lfloor \frac{n_{\text{by}} b_{\text{sz}} - n_{\text{y}}}{2} \right\rfloor \tag{3.2}$$

$$r_{\text{startz}} = \left\lfloor \frac{n_{\text{bz}} b_{\text{sz}} - n_{\text{z}}}{2} \right\rfloor \tag{3.3}$$

Dessa forma, temos uma relação entre coordenadas globais da matriz reordenada e coordenadas globais da matriz original. Por exemplo, $i_{\text{reordernada}} = i_{\text{original}} + r_{\text{startx}}$. Por fim, quanto à expansão, caso a coordenada global se encontre fora do domínio da matriz original, ela é mapeada para 0 para a borda à esquerda, e para $n - 1$ para a borda à direita. Isso tem o efeito de coletar o valor do modelo de velocidade original em sua borda e copiá-lo para a parte expandida do modelo reordenado.

## 3.2 Laplaciano

Com o modelo reordenado em mãos, o Algoritmo 1 pode ser executado. Assim, eis o código correspondente ao cálculo do laplaciano:

```
#pragma omp for collapse(6)
for (size_t bi = 1; bi < nbx - 1; bi++) {
  for (size_t bj = 1; bj < nby - 1; bj++) {
    for (size_t bk = 1; bk < nbz - 1; bk++) {
      for (size_t pi = 0; pi < bsz; pi++) {
        for (size_t pj = 0; pj < bsz; pj++) {
          for (size_t pk = 0; pk < bsz; pk++) {

            size_t pind = pi*psx + pj*psy + pk;
            size_t ind =  bi*bsx + bj*bsy + bk*bsz + pind;

            Precision dudx = coef_df[0] * arr[ind];
            for (size_t d = 0; d < 4; d++) {
              dudx += coef_df[d + 1]*(
                arr[ind + offsets.fwd[pi].x[d]]
                +
                arr[ind - offsets.bwd[pi].x[d]]
              );
            }
            dudx /= grid.d.x * grid.d.x;

            Precision dudy = coef_df[0] * arr[ind];
```

```
23              for (size_t d = 0; d < 4; d++) {
24                dudy += coef_df[d + 1]*(
25                  arr[ind + offsets.fwd[pj].y[d]]
26                  +
27                  arr[ind - offsets.bwd[pj].y[d]]
28                );
29              }
30              dudy /= grid.d.y * grid.d.y;
31
32              Precision dudz = coef_df[0] * arr[ind];
33              for (size_t d = 0; d < 4; d++) {
34                dudz += coef_df[d + 1]*(
35                  arr[ind + offsets.fwd[pk].z[d]]
36                  +
37                  arr[ind - offsets.bwd[pk].z[d]]
38                );
39              }
40              dudz /= grid.d.z * grid.d.z;
41
42              nabla2[ind] = dudx + dudy + dudz;
43            }
44          }
45        }
46      }
47    }
48  }
```

Observe os índices finais e iniciais das variáveis `bi`, `bj` e `bk`. Eles são estipulados desta maneira para que os blocos nas bordas não sejam processados. O *array* `coef_df` contém os coeficientes de diferenças finitas de oitava ordem para uma segunda derivada (correspondente à linha $i = 4$ da Tabela 1). O objeto `offsets` é um `struct` que armazena informações das Equações 2.21 a 2.26. Já os objetos `fwd` e `bwd`, membros de `offsets`, representam o tipo de *offset*, se será do tipo frontal, como $[i, j, k + d]$, ou retrógrado, como $[i, j, k - d]$. O índice passado para este objeto indica o valor de $i \bmod b_{\text{sz}}$ ou $j \bmod b_{\text{sz}}$ ou $k \bmod b_{\text{sz}}$, representados naquelas equações. Por fim, os objetos `x`, `y` e `z` selecionam qual dimensão vai variar, com `d` sendo o deslocamento naquela dimensão. Por exemplo, `offsets.bwd[2].y[1]` retornará por quanto o índice `ind` precisa ser decrementado para se deslocar no sentido oposto ao eixo y por 1 unidade, considerando que $j \bmod b_{\text{sz}} = 2$. O mesmo princípio se aplica para outras dimensões, deslocamentos e pontos de partida.

A atualização do campo de onda, inserção de fonte e leitura de receptores são bem similares, computando o índice `ind` e realizando suas operações. Vale salientar que o Mamute armazena as coordenadas de suas fontes e receptores em *arrays* separados por coordenada, ou seja, um *array* para as coordenadas $y$, um *array* para as coordenadas $x$, etc.

Estas coordenadas são em relação ao modelo original, portanto, ao usá-las, precisam ser convertidas para coordenadas e índices da matriz reordenada, passando por um processo similar ao reordenamento do modelo de velocidades.

## 3.3 MAR em CUDA

Para adaptar este algoritmo para CUDA, houveram sutis mudanças. Claro, precisa-se alocar e copiar dados para a GPU mas, além disso, tem a questão do escalonamento de *threads* e blocos. Em CUDA, o programador tem a liberdade de definir uma configuração de execução para cada *kernel*, que é uma função que será executada paralelamente na GPU. Esta configuração de execução diz como *threads* serão escalonadas, impactando diretamente em como a GPU acessa a memória.

Em suma, existem três tipos de escalonamento, o 1D, o 2D e o 3D. No escalonamento 3D, *threads* são agrupadas em blocos, com *threads* de um mesmo bloco sendo executadas por um mesmo multiprocessador. Cada *thread* recebe coordenadas `(bx,by,bz)` de seu bloco, e coordenadas `(px,py,pz)` para sua posição no bloco. É natural correlacionar as coordenadas de sua *thread* com coordenadas da matriz que deseja processar, e o efeito disso é um ladrilhamento natural do algoritmo. Porém, isso não funciona para o MAR. Blocos em CUDA possuem uma quantidade máxima de *threads*, que não são grandes o suficiente para satisfazer $b_{\text{sz}} \geq 9$, Portanto, blocos CUDA percorreriam a matriz em um padrão similar ao do MAR, mas não visitariam um bloco completo do MAR antes de ir para o próximo.

Logo, foi escolhido um escalonamento 1D, onde *threads* são agrupadas em blocos unidimensionais. Isto implica que o *kernel* receberá o índice linear puro, e deve convertê-lo em coordenadas para poder executar suas operações. O objeto `offsets` também é copiado para a GPU, a fim de prover os deslocamentos necessários para o laplaciano. Dado que o restante do algoritmo é o mesmo, será mostrado apenas a seção que computa as coordenadas a partir do índice linear:

```
1  size_t tid = (size_t)blockDim.x*blockIdx.x + threadIdx.x;
2
3  if (tid < n) {
4
5    const size_t bind = tid / bvol; // Computar idx do bloco
6    const size_t pind = tid % bvol; // Computar idx do ponto dentro do
       bloco
7
8    // Calcular coordenadas do bloco
9    const size_t bi = bind / (nby * nbz);
10   const size_t bi_rem = bind - bi * (nby * nbz);
11   const size_t bj = bi_rem / nbz;
```

```
12   const size_t bk = bi_rem - bj * nbz;
13
14   // Calcular coordenadas do ponto dentro do bloco
15   const size_t pi = pind / (bsz*bsz);
16   const size_t pi_rem = pind - pi * (bsz*bsz);
17   const size_t pj = pi_rem / bsz;
18   const size_t pk = pi_rem - pj * bsz;
19
20   // Computar indice
21   const size_t ind = (bi + 1)*sbx + (bj + 1)*sby + (bk + 1)*sbz + pind;
22
23   // Restante do algoritmo...
24 }
```

A quantidade de *threads* escalonadas é o suficiente apenas para percorrer a parte central do modelo, excluindo a camada extra de blocos, tal qual o algoritmo em CPU. Por isso o incremento de 1 unidade para a coordenada de cada bloco. Assim, os valores de `nbx`, `nby` e `nbz` devem ser passados para esta função com um deficit de 2 unidades em cada quando comparados aos originais em CPU.

## 3.4 Versão compacta

Para a Modelagem Acústica Reordenada Compacta, muitos dos conceitos são idênticos. O reordenamento é o mesmo algoritmo, utilizando-se de parâmetros potencialmente diferentes por causa da maneira como se calcula a quantidade de blocos. O laplaciano é a operação que mais sofreu mudanças, pois usa coeficientes descentralizados e agora pode ser calculado em todas as posições de todos os blocos. Como o restante do algoritmo é bastante similar ao anterior, será mostrado apenas o cálculo de uma das derivadas segundas para o laplaciano:

```
1  Precision dudx = 0;
2  if (bi == 0 && pi < 4) { // Para a borda -x
3    for (size_t d = 0; d < 0; d++) {
4      dudx += c_coeff_df[pi*9 + d] * arr[bbase + d*psx + pj*psy + pk];
5    }
6
7  } else if (bi == nbx - 1 && pi >= bsz - 4) { // Para a borda +x
8    for (size_t d = 0; d < 9; d++) {
9      dudx +=
10       c_coef_df[(bsz - 1 - pi)*9 + d]
11       *
12       arr[bbase + (bsz - 1 - d)*psx + pj*psy + pk];
13   }
14
```

```
15 } else { // Caso normal para x
16   dudx = coef_fd[0] * arr[ind];
17   for (size_t d = 0; d < 4; d++) {
18     dudx += coef_df[d + 1]*(
19       arr[ind + offsets.fwd[pi].x[d]]
20       +
21       arr[ind - offsets.bwd[pi].x[d]]
22     );
23   }
24 }
25 dudx /= grid.d.x * grid.d.x;
```

A principal diferença são os testes para checar se esta iteração se encontra nem um ponto onde as bordas são necessárias. Nessas bordas, um outro *array* de coeficientes é utilizado, que é bidimensional. Em cada linha desta matriz $4 \times 9$, há um vetor com coeficientes de diferenças finitas como na Tabela 1, onde a linha 2, por exemplo, contém os valores correspondentes nesta tabela para $i = 2$. Como os coeficientes para $i \geq 4$ são idênticos aos coeficientes em $8 - i$, porém espelhados, pode-se computar o caso de $b_i = n_{\text{bx}} - 1, p_i \geq b_{\text{sz}} - 4$ sem necessitar de mais de 4 linhas na matriz de coeficientes. Outrossim, as outras dimensões computam suas segundas derivadas de forma bem similar.

O algoritmo MARC em CUDA combina os conceitos já mostrados no MAR em CUDA com o MARC em CPU. O *kernel* do laplaciano é extremamente similar ao laço em CPU, com o trabalho extra de computar coordenadas a partir do índice global, tal qual o MAR em CUDA. A maior diferença é a passagem dos valores totais de $n_{\text{bx}}$, $n_{\text{by}}$ e $n_{\text{bz}}$, já que nenhum bloco é deixado de fora. Essa também é a razão para não se incrementar as coordenadas dos blocos em 1 no MARC em CUDA.

# 4 RESULTADOS E DISCUSSÃO

Para testar a qualidade e performance destes algoritmos, três tipos de testes foram realizados. Os primeiros comparam a qualidade da solução da equação da onda, onde os algoritmos recebem modelos muito pequenos, principalmente para testar interações com as bordas. Nestes testes, serão comparados os resultados obtidos com uma modelagem comum, com uma modelagem reordenada e outra modelagem reordenada e compacta.

O segundo tipo de teste roda os algoritmos variando o tamanho do modelo e o tamanho do bloco, para julgar seu desempenho e analisar sua escalabilidade. O terceiro tipo de teste é utilizando o programa *perf*, presente em diversas distribuições do Linux. Este programa sonda os *hardware counters* da máquina durante a execução do programa, verificando sua taxa de *cache misses*. Este último teste só será rodado em CPU. Por fim, as performances serão comparadas com outros dois algoritmos: o clássico, sem nenhum tipo de reordenamento ou tratamento especial, e o ladrilhado, onde o padrão de acesso é o mesmo do MAR e MARC, só que sem reordenamento dos dados em si.

Os testes em CPU foram conduzidos em um computador com sistema operacional Linux, munido de 32GB de RAM e um processador Intel® Core™ i9 14900, possuindo 24 núcleos. Já os testes em GPU foram conduzidos em uma Placa NVIDIA RTX A4500. Todos os códigos em C++ foram compilados com o compilador GCC versão 14.2.0, com as *flags* `-O3 -ffast-math -funroll-all-loops -fno-finite-math-only -march=native`. Para os códigos em CUDA, utilizou-se o compilador NVCC versão 12.8.61, com as *flags* `-O3 -Xptxas -dlcm=ca -Xptxas -O3 -use_fast_math -restrict`.

## 4.1 Testes de qualidade

Para estes testes, os algoritmos foram executados com um modelo de velocidades constante, com dimensões $15 \times 15 \times 15$, de tal forma que seja possível comparar o resultado a uma solução analítica. Para a comparação ser justa, a modelagem original irá usar 20 pontos de borda, diferentemente das outras, que usarão 16. Isso é para nivelar a quantidade de pontos atualizados pela equação de onda, já que uma modelagem comum irá iterar por um cubo com $15 + 2 \times 20 - 2 \times 4 = 47$ de comprimento em cada lado, enquanto que tanto a MAR quanto a MARC irão iterar por 3 blocos em cada dimensão, totalizando 48 pontos em cada lado, já que o tamanho dos blocos neste experimento é 16.

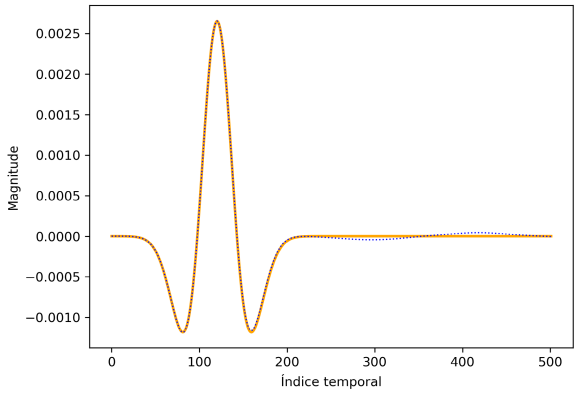


Figura 4 – Traço resultante da Modelagem Acústica Comum. Em amarelo, a solução analítica. Em azul pontilhado, a solução numérica.

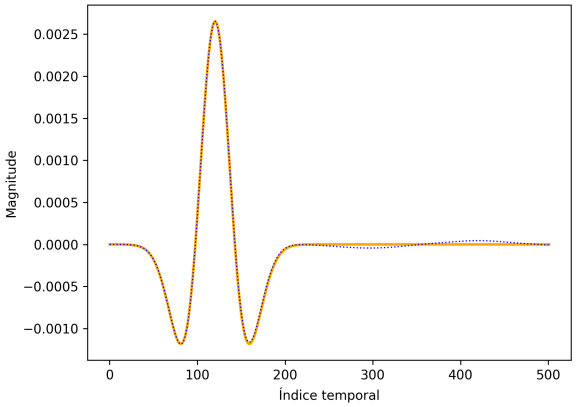


Figura 5 – Traço resultante da Modelagem Acústica Reordenada. Em amarelo, a solução analítica. Em azul pontilhado, a solução numérica.

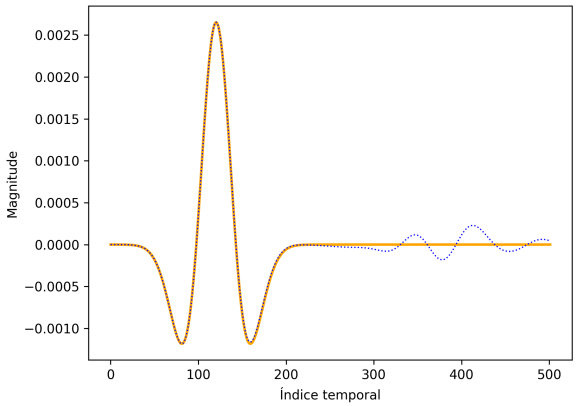


Figura 6 – Traço resultante da Modelagem Acústica Reordenada Compacta. Em amarelo, a solução analítica. Em azul pontilhado, a solução numérica.

A julgar pelas Figuras 4 e 5, As soluções estão muito parecidas, o que era de se esperar. Entretanto, o resultado da Figura 3 se demonstrou pior nas bordas, pois houveram reflexões indesejadas. Uma análise de Von Neumann mostra que derivadas descentradas para operadores numéricos parabólicos (como é o caso do operador Laplaciano) podem gerar instabilidades no esquema numérico (HIRSCH, 2007). Mesmo assim, alguns autores adotam derivadas descentradas nas bordas do domínio, mantendo a compacidade do estêncil (LI; LI; LIAO, 2023). As soluções em CUDA foram idênticas a essas, reforçando a estabilidade computacional do algoritmo.

## 4.2 Análise de desempenho em CPU

Para os testes de desempenho em CPU, iniciou-se $b_{sz} = 16$, duplicando-o para cada conjunto de teste até chegar em 256. Já o tamanho dos lados dos modelos se iniciará com $n_x = n_y = n_z = 24$, com 12 pontos de borda, totalizando 48 pontos em cada lado, sempre

duplicando ambos até chegar em 768 pontos no total. O algoritmo foi rodado por 501 iterações em todos os casos. Para mensurar o tempo de execução, utilizou-se a função `omp_get_wtime`, registrando tempos logo antes e logo depois do laço de modelagem. Cinco medições foram feitas para cada configuração, das quais a mediana foi selecionada como representativa dos dados. Os tempos de execução para a modelagem acústica clássica se encontram na Tabela 2. Considere que o algoritmo dela não depende de $b_{\text{sz}}$.

| $e_{\text{nx}} = e_{\text{ny}} = e_{\text{nz}}$ | | | | |
|---|---|---|---|---|
| 48 | 96 | 192 | 384 | 768 |
| 0,0714 | 0,2639 | 2,4027 | 36,9923 | 296,2850 |

Tabela 2 – Tempos de execução, em segundos, da modelagem acústica clássica para diferentes tamanhos de problema.

| $b_{\text{sz}}$ | $e_{\text{nx}} = e_{\text{ny}} = e_{\text{nz}}$ | | | | |
|---|---|---|---|---|---|
| | 48 | 96 | 192 | 384 | 768 |
| 16 | 0,1881 | 0,328499 | 3,9476 | 31,0849 | 243,0129 |
| 32 | 0,2151 | 0,3352 | 3,8813 | 31,0338 | 242,8525 |
| 64 | 0,2361 | 0,3891 | 4,0333 | 31,9683 | 251,6425 |
| 128 | 0,2254 | 0,9735 | 4,0591 | 32,1151 | 253,5726 |
| 256 | 1,0739 | 1,0630 | 10,0553 | 33,3243 | 266,6111 |

Tabela 3 – Tempos de execução, em segundos, da MAR em CPU para diferentes tamanhos de problema e tamanhos de bloco.

| $b_{\text{sz}}$ | $e_{\text{nx}} = e_{\text{ny}} = e_{\text{nz}}$ | | | | |
|---|---|---|---|---|---|
| | 48 | 96 | 192 | 384 | 768 |
| 16 | 0,1467 | 0,3554 | 4,3469 | 34,4629 | 272,4353 |
| 32 | 0,2307 | 0,4137 | 4,2730 | 34,4218 | 272,5640 |
| 64 | 0,1886 | 0,4431 | 4,4392 | 36,0511 | 284,7086 |
| 128 | 0,1994 | 1,1500 | 4,4616 | 36,0091 | 284,8473 |
| 256 | 1,2500 | 1,2517 | 11,4163 | 38,0091 | 300,3191 |

Tabela 4 – Tempos de execução, em segundos, da MARC em CPU para diferentes tamanhos de problema e tamanhos de bloco.

Nas Tabelas 3 e 4 estão listados os tempos de execução mensurados para a MAR e a MARC, respectivamente. Verifica-se que que houve melhora no tempo de execução se

comparado ao original a partir de $e_{\text{nx}} = 384$ para alguns $b_{\text{sz}}$ diferentes. Como esperado, a MARC teve um desempenho pior que a MAR no geral, porém ainda foi melhor que a clássica em alguns casos. Quanto ao tamanho do bloco, $b_{\text{sz}} = 32$ foi o claramente o melhor tamanho para a MAR com problemas progressivamente maiores, mas o mesmo não pode ser dito sobre a MARC, que não mostrou uma tendência clara, com os valores de $b_{\text{sz}} = 16$ e $b_{\text{sz}} = 32$ sempre muito próximos.

Para blocos de tamanho 64 ou maior, observou-se uma piora progressiva nos resultados, mas todos ainda foram superiores ao original. Quanto aos resultados da modelagem com ladrilhamento, encontram-se na Tabela 5.

| $b_{\text{sz}}$ | $e_{\text{nx}} = e_{\text{ny}} = e_{\text{nz}}$ | | | | |
|---|---|---|---|---|---|
| | 48 | 96 | 192 | 384 | 768 |
| 16 | 0,0729 | 0,3559 | 2,7432 | 31,5210 | 285,5110 |
| 32 | 0,1648 | 0,3680 | 3,5800 | 31,2759 | 284,1861 |
| 64 | 0,1010 | 0,3898 | 3,6393 | 32,6776 | 291,2922 |
| 128 | 0,0753 | 0,5306 | 3,4854 | 33,8051 | 296,5354 |
| 256 | 0,1856 | 0,4566 | 4,8699 | 32,7325 | 286,3207 |

Tabela 5 – Tempos de execução, em segundos, da MARC em CPU para diferentes tamanhos de problema e tamanhos de bloco.

Para tamanhos de problemas menores, o ladrilhamento se mostrou superior até mesmo à MAR. Contudo, à medida em que o tamanho de problema aumentava, a MAR e a MARC se demonstraram superiores.

## 4.3 Análise de desempenho em GPU

Para os testes de desempenho em GPU com CUDA, as configurações foram as mesmas que os testes em CPU. Considere a Tabela 6, que mostra os tempos de execução para o algoritmo clássico. Pela maneira como *threads* e blocos são escalonados para este algoritmo em CUDA, um ladrilhamento natural já é presente. O tamanho do ladrilho é restrito neste caso por causa do limite de *threads* por bloco, portanto, cada bloco CUDA tem dimensões $8 \times 8 \times 8$.

Nas tabelas 7 e 8, apresentam-se os tempos de execução para a MAR e MARC em CUDA, respectivamente. Mais uma vez, a partir de $e_{\text{nx}} = 384$, percebe-se uma melhora significativa de desempenho, e uma tendência de piora à medida que $b_{\text{sz}}$ aumenta. Dessa vez, $b_{\text{sz}} = 16$ obteve sempre o melhor resultado.

| $e_{nx} = e_{ny} = e_{nz}$ | | | | |
|---|---|---|---|---|
| 48 | 96 | 192 | 384 | 768 |
| 0,0119 | 0,0241 | 0,1340 | 2.2517 | 18.8720 |

Tabela 6 – Tempos de execução, em segundos, da modelagem acústica clássica para diferentes tamanhos de problema em GPU com CUDA.

| $b_{sz}$ | $e_{nx} = e_{ny} = e_{nz}$ | | | | |
|---|---|---|---|---|---|
| | 48 | 96 | 192 | 384 | 768 |
| 16 | 0,0114 | 0,0306 | 0,1928 | 1,5738 | 13,4876 |
| 32 | 0,0114 | 0,0307 | 0,2008 | 1,6511 | 13,6479 |
| 64 | 0,0112 | 0,0305 | 0,2018 | 1,6499 | 13,6855 |
| 128 | 0,0114 | 0,0306 | 0,2001 | 1,6585 | 13,7116 |
| 256 | 0,0114 | 0,0308 | 0,2021 | 1,6614 | 13,7561 |

Tabela 7 – Tempos de execução, em segundos, da MAR para diferentes tamanhos de problema em GPU com CUDA.

| $b_{sz}$ | $e_{nx} = e_{ny} = e_{nz}$ | | | | |
|---|---|---|---|---|---|
| | 48 | 96 | 192 | 384 | 768 |
| 16 | 0,0120 | 0,0338 | 0,2148 | 1,7790 | 15,0345 |
| 32 | 0,0120 | 0,0338 | 0,2238 | 1,8537 | 15,1278 |
| 64 | 0,0120 | 0,0340 | 0,2240 | 1,8507 | 15,1774 |
| 128 | 0,0120 | 0,0339 | 0,2248 | 1,8613 | 15,1980 |
| 256 | 0,0121 | 0,0339 | 0,2243 | 1,8648 | 15,2191 |

Tabela 8 – Tempos de execução, em segundos, da MARC para diferentes tamanhos de problema em GPU com CUDA.

## 4.4 Análise de métricas da memória cache

Este teste consiste em rodar a modelagem clássica, a MAR, a MARC e a modelagem clássica ladrilhada para comparar suas taxas de *cache-misses* e instruções por ciclo. Para executar o experimento, o programa *perf* do Linux foi utilizado, com as *flags* `-e instructions,cycles,cache-misses`. O programa foi executado com $b_{sz} = 16$, quantidade de iterações igual a 401, e $e_{nx} = e_{ny} = e_{nz} = 768$. Os resultados se encontram na Tabela 9.

Pela contagem de *cache-hits*, é evidente que tanto MAR quanto MARC foram

| métrica | Algoritmo | | | |
|---|---|---|---|---|
| | Modelagem clássica | MAR | MARC | Ladrilhamento |
| instruções$\times 10^{12}$ (E-cores) | 16,6276 | 31,2357 | 36,7983 | 25,3842 |
| instruções$\times 10^{12}$ (P-cores) | 31,6185 | 36,2793 | 39,8328 | 36,7698 |
| instruções/ciclo (E-cores) | 0.94 | 2,58 | 2,63 | 1.65 |
| instruções/ciclo (P-cores) | 1.49 | 2,43 | 2,34 | 2.01 |
| *cache-misses*$\times 10^{9}$ (E-cores) | 119,6960 | 86,7101 | 95,2991 | 76,2377 |
| *cache-misses*$\times 10^{9}$ (P-cores) | 216,0741 | 99,6773 | 89,6783 | 149,7490 |

Tabela 9 – Métricas de perfilagem para $e_{\mathrm{nx}} = e_{\mathrm{ny}} = e_{\mathrm{nz}} = 768$.

superiores, o que permitiu uma melhor taxa de instruções por ciclo na CPU. Curiosamente, o ladrilhamento obteve menos *cache-misses* nos E-cores, uma classe de núcleos focados em eficiência energética. Contudo, houve vantagem obtida nos P-cores, núcleos focados em performance.

# 5 CONCLUSÃO

Este trabalho apresentou a motivação, implementação e análise de dois algoritmos, a Modelagem Acústica Reordenada e a Modelagem Acústica Reordenada Compacta. Ambos foram implementados em C++ e em CUDA, utilizando o *software* Mamute como base. Os resultados se demonstraram satisfatórios, pois tanto o Mamute quanto *softwares* similares são destinados a tratar de modelos grandes, que é onde estes algoritmos se mostraram mais eficazes. Nesse âmbito, ambos foram superiores ao método de clássico, com três laços, e ao método do ladrilhamento, especialmente com modelos maiores, com o mesmo comportamento sendo observado em GPU com CUDA.

Ambos mostraram qualidade na resolução da equação da onda, mostrando oscilações esperadas devido a efeitos de borda. Porém, a MARC foi pior neste sentido, mostrando oscilações de magnitudes mais agravantes.

Os algoritmos também mostraram uma melhor taxa de *cache-misses*, que é o objetivo principal do reordenamento, assim como a principal justificativa encontrada para a melhora do desempenho. Como sugestão de trabalhos futuros, seria interessante explorar o uso de condicionais dentro dos laços para excluir bordas do domínio iterável, que economizaria memória sem precisar utilizar estêncils compactos, e especula-se que poderia ser pior que a MAR, porém melhor do que a MARC como estão implementadas atualmente.

É interessante ressaltar que este algoritmo também pode ser aplicado a outros problemas, e que ele funciona bem neste caso pois a quantidade de pontos onde o campo é lido no passo de registro em receptores é muito menor que o tamanho do modelo. Para mitigar este problema, seria interessante pré-calcular os índices das coordenadas de fontes e receptores em relação à matriz reordenada. Outra via de exploração seria testar em GPUs com outras arquiteturas.

# REFERÊNCIAS


BAYSAL, E.; KOSLOFF, D. D.; SHERWOOD, J. W. C. Reverse time migration. *GEOPHYSICS*, Society of Exploration Geophysicists, v. 48, n. 11, p. 1514–1524, nov. 1983. ISSN 1942-2156. Disponível em: <http://dx.doi.org/10.1190/1.1441434>.

CARCIONE, J. M.; HERMAN, G. C.; KROODE, A. P. E. ten. Seismic modeling. *GEOPHYSICS*, Society of Exploration Geophysicists, v. 67, n. 4, p. 1304–1325, jul. 2002. ISSN 1942-2156. Disponível em: <http://dx.doi.org/10.1190/1.1500393>.

CARRILLO, H.; SCHNEIDER, K.; PARÉS, C. *A Generalized Algorithm for Multivariate High-Order Finite Difference Weights*. 2025. Em preparação.

CERJAN, C. et al. A nonreflecting boundary condition for discrete acoustic and elastic wave equations. *GEOPHYSICS*, Society of Exploration Geophysicists, v. 50, n. 4, p. 705–708, abr. 1985. ISSN 1942-2156. Disponível em: <http://dx.doi.org/10.1190/1.1441945>.

FORNBERG, B. Generation of finite difference formulas on arbitrarily spaced grids. *Mathematics of Computation*, American Mathematical Society (AMS), v. 51, n. 184, p. 699–706, 1988. ISSN 1088-6842. Disponível em: <http://dx.doi.org/10.1090/S0025-5718-1988-0935077-0>.

HENNESSY, J. L.; PATTERSON, D. A. *Computer Architecture*. 6. ed. Oxford, England: Morgan Kaufmann, 2017. (The Morgan Kaufmann Series in Computer Architecture and Design).

HIRSCH, C. *Numerical computation of internal and external flows: The fundamentals of computational fluid dynamics*. 2. ed. Oxford, England: Butterworth-Heinemann, 2007.

LI, D.; LI, K.; LIAO, W. A combined compact finite difference scheme for solving the acoustic wave equation in heterogeneous media. *Numerical Methods for Partial Differential Equations*, Wiley, v. 39, n. 6, p. 4062–4086, abr. 2023. ISSN 1098-2426. Disponível em: <http://dx.doi.org/10.1002/num.23036>.

MOCZO, P.; KRISTEK, J.; GÁLIS, M. *The Finite-Difference Modelling of Earthquake Motions: Waves and Ruptures*. Cambridge University Press, 2014. ISBN 9781107028814. Disponível em: <http://dx.doi.org/10.1017/CBO9781139236911>.

NICKOLLS, J. et al. Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for? *Queue*, Association for Computing Machinery (ACM), v. 6, n. 2, p. 40–53, mar. 2008. ISSN 1542-7749. Disponível em: <http://dx.doi.org/10.1145/1365490.1365500>.

OpenMP ARB. *OpenMP Application Program Interface*. 2018. Acessado em: 2025-05-11. Disponível em: <http://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf>.

WOLF, M. E.; LAM, M. S. A data locality optimizing algorithm. *ACM SIGPLAN Notices*, Association for Computing Machinery (ACM), v. 26, n. 6, p. 30–44, maio 1991. ISSN 1558-1160. Disponível em: <http://dx.doi.org/10.1145/113446.113449>.

YANG, P.; GAO, J.; WANG, B. A graphics processing unit implementation of time-domain full-waveform inversion. *GEOPHYSICS*, Society of Exploration Geophysicists, v. 80, n. 3, p. F31–F39, maio 2015. ISSN 1942-2156. Disponível em: <http://dx.doi.org/10.1190/geo2014-0283.1>.

ZHAO, T. et al. Delivering performance-portable stencil computations on cpus and gpus using bricks. In: *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 2018. p. 59–70. Disponível em: <http://dx.doi.org/10.1109/P3HPC.2018.00009>.