UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

INTEGRANDO CHECKPOINTING E COMPRESSÃO DE DADOS DE UM SOLVER FWI COM OPENMP EM MULTI-GPUS

Yuri Nicolau Freire

São Carlos
2025

YURI NICOLAU FREIRE

INTEGRANDO CHECKPOINTING E COMPRESSÃO DE DADOS DE UM SOLVER FWI COM OPENMP EM MULTI-GPUS

Trabalho de conclusão de curso apresentado como requisito parcial para a obtenção do título de Bacharel em Engenharia da Computação pela Universidade Federal de São Carlos.

Orientador: Prof. Dr. Hermes Senger.

São Carlos

2025

FREIRE, Yuri Nicolau.
Nº Cutter Integrando checkpointing e compressão de dados de um solver FWI com OpenMP em multi-GPUs.. — 2025. [qtd. de folhas] f.

Trabalho de Conclusão de Curso (Bacharel em Engenharia da Computação) – Universidade Federal de São Carlos, São Carlos, 2025.

1. [primeira entrada de assunto]. 2. [segunda entrada de assunto]. 3. [terceira entrada de assunto]. I. Título.

CDD [número da CDD].

Integrando checkpointing e compressão de dados de um solver FWI com OpenMP em multi-GPUs.

Yuri Nicolau Freire

Trabalho de conclusão de curso apresentado como requisito parcial para a obtenção do título de Bacharel em Engenharia da Computação pela Universidade Federal de São Carlos.

Aprovado em: 25/02/2025.

BANCA EXAMINADORA

_____

Orientador

Prof. Dr. Hermes Senger

Universidade Federal de São Carlos

_____

Membro da banca (1)

Prof. Dr. Hélio Crestana Guardia

Universidade Federal de São Carlos

_____

Membro da banca (2)

Prof. Dr. Ricardo Menotti

Universidade Federal de São Carlos

À minha mãe, meu porto seguro

AGRADECIMENTOS

Agradecemos pelo apoio recebido das seguintes agências de fomento:

- Fundação de Amparo à Pesquisa do Estado de São Paulo (Processos 2019/26702-8, 2021/00199-8 e 2023/00566-6).
- Conselho Nacional de Pesquisa e Desenvolvimento-CNPq (Processo 302296/2023-9);
- Financiadora de Estudos e Projetos - FINEP / Ministério da Ciência, Tecnologia e Inovações - MCTI / Fundo Nacional de Desenvolvimento Científico e Tecnológico – FNDC (Convênio 01.23.0575.00 - 0381/23);

Este trabalho, assim como toda a minha graduação, só foi possível graças ao apoio de familiares, amigos e docentes, que me acompanharam e auxiliaram em diferentes momentos ao longo dos últimos anos.

Primeiramente, agradeço à minha família, em especial à minha mãe, Elaine, que, desde o início, me incentivou a correr atrás dos meus sonhos. Desde a escolha de um curso que parecia impossível em um primeiro momento, passando pela mudança para Uberlândia e, posteriormente, para São Carlos, quando percebi que a instituição inicial não atenderia minhas expectativas, até os momentos de dúvida e vontade de desistir, ela esteve sempre presente, me apoiando e aconselhando da melhor maneira possível.

Não poderia deixar de agradecer também à minha irmã, que sempre serviu de guia para mim. Sem seguir seus passos desde a infância, tenho certeza de que minha trajetória teria sido muito diferente. Como luzes na neblina, suas sábias escolhas e conselhos foram fundamentais para o meu caminho.

Agradeço ainda ao meu pai, que nunca deixou de me apoiar sempre que necessário, e às minhas primas e minha tia, que reforçam a sensação de "estar em casa" não por um ponto no mapa, mas pela companhia.

Além do vínculo familiar, sou grato à minha namorada, Juliana, que esteve ao meu lado na fase final da graduação, me auxiliou nos trabalhos acadêmicos e, junto aos nossos dois gatos, enche nossa casa de vida. Agradeço também aos meus amigos da faculdade; sem eles, os anos de graduação pareceriam décadas, e não meses.

Por fim, agradeço ao professor Hermes, que me orientou ao longo do meu trabalho de iniciação científica, auxiliando na escrita de artigos e apresentações em

eventos da área de HPC; à Capes, pelo financiamento por meio de bolsa durante esse período; e à banca examinadora, pelo tempo dedicado à leitura deste trabalho, pelas correções e sugestões de melhoria, e pela atenção à sua apresentação.

RESUMO

A computação de alto desempenho (HPC) desempenha um papel crucial em diversas áreas científicas, e o uso de GPUs para computação paralela tem se mostrado uma solução eficiente para problemas de grande escala, como a inversão de forma de onda completa (FWI) em simulações sísmicas. Este trabalho propõe a implementação de técnicas de decomposição de domínio, *checkpointing* e compressão de dados em uma aplicação FWI executada em múltiplas GPUs, com o objetivo de otimizar o desempenho computacional e reduzir o uso de memória. A pesquisa foi realizada em três etapas principais: a familiarização com a base de código existente, o estudo das técnicas de *checkpointing* e compressão de dados, e a implementação da decomposição de domínio em um ambiente multi-GPU. A estratégia de decomposição de domínio foi aplicada por meio da subdivisão do grid tridimensional em "fatias", permitindo a execução distribuída entre múltiplos dispositivos. O controle explícito da alocação e transferência de dados entre a CPU e as GPUs, utilizando OpenMP e nvcomp, possibilitou uma maior eficiência na gestão da memória e nas transferências de dados. Os resultados experimentais, realizados em um nó com 4 GPUs NVidia V100 conectadas via NVLink, mostraram que a utilização de múltiplas GPUs oferece ganhos significativos de desempenho, principalmente para problemas de grande escala. Para problemas menores, o *overhead* associado à transferência de dados entre dispositivos neutraliza os benefícios do paralelismo. A combinação de checkpointing e compressão de dados resultou em melhorias no uso de memória e na comunicação entre dispositivos, destacando-se especialmente em domínios maiores e com maior número de iterações. Este trabalho contribui para o avanço das aplicações de FWI em ambientes de computação distribuída, oferecendo uma solução escalável e eficiente para problemas científicos de grande porte.

Palavras-chave: FWI, CUDA, OpenMP, Descomposição de Domínio.

ABSTRACT

High-performance computing (HPC) plays a crucial role in various scientific fields, and the use of GPUs for parallel computing has proven to be an efficient solution for large-scale problems, such as Full Waveform Inversion (FWI) in seismic simulations. This work proposes the implementation of domain decomposition, checkpointing, and data compression techniques in a FWI application executed on multiple GPUs, aiming to optimize computational performance and reduce memory usage. The research was conducted in three main stages: familiarization with the existing codebase, study of checkpointing and data compression techniques, and the implementation of domain decomposition in a multi-GPU environment. The domain decomposition strategy was applied through the subdivision of the three-dimensional grid into "slices," allowing distributed execution across multiple devices. Explicit control over data allocation and transfer between the CPU and GPUs, using OpenMP and nvcomp, enabled greater efficiency in memory management and data transfers. Experimental results, performed on a node with 4 NVidia V100 GPUs connected via NVLink, showed that the use of multiple GPUs offers significant performance gains for large-scale problems. For smaller problems, the overhead associated with data transfer between devices neutralizes the benefits of parallelism. The combination of checkpointing and data compression resulted in improvements in memory usage and communication between devices, particularly in larger domains and with more iterations. This work contributes to the advancement of FWI applications in distributed computing environments, providing a scalable and efficient solution for large-scale scientific problems.

Keywords: FWI, CUDA, OpenMP, Domain Decomposition.

LISTA DE FIGURAS

LISTA DE ALGORITMOS

LISTA DE TABELAS

LISTA DE SIGLAS

- FWI - Full Waveform Inversion
- GPU - Graphics Processing Unit
- CPU - Central Processing
- HPC - High Performance Computing
- EDP - Equação Diferencial Parcial
- PCIe - Peripheral Component Interconnect Express
- UVA - Unified Virtual Addressing

SUMÁRIO

1 INTRODUÇÃO

A computação de alto desempenho (*High Performance Computing* - HPC) tem sido uma ferramenta muito importante para o avanço de inúmeras áreas científicas, permitindo que simulações e cálculos em grandes escalas, como simulação de sistemas gravitacionais com muitas partículas, física atmosférica, projetos de física com partículas de alta energia, entre outros, sejam realizados (Ponce *et al*., 2019). Com o crescimento da demanda por processamento mais rápido e eficiente, o uso de técnicas de paralelismo se tornou imprescindível para maximizar o desempenho computacional oferecido pelos supercomputadores modernos.

Historicamente, o aumento no poder de processamento se alinhava à Lei de Moore, que tinha como hipótese crescimento dobrado do número de transistores das CPUs a cada 2 anos (Moore, 1965). Os avanços recentes em *hardware*, contudo, têm desacelerado essa tendência, exigindo abordagens inovadoras que vão além do aumento na densidade dos transistores. Dentre essas abordagens, o uso das centenas a milhares de núcleos de processamento disponíveis em GPUs se destaca como uma das mais impactantes (Emel'yanov *et al.*, 2017; Serpa *et al.*, 2021; Pandey *et al.*, 2022).

As GPUs, originalmente projetadas para processamento gráfico, evoluíram de forma que hoje são ferramentas poderosas para computação paralela geral (Pandey *et al.*, 2022). Por conta do grande número de núcleos de processamento em um único *chip*, as GPUs oferecem um ambiente ideal para a execução de aplicações altamente paralelizáveis. Isso permite que problemas computacionais intensivos, como os encontrados em simulações científicas e engenharia, sejam resolvidos com ganhos significativos de desempenho (Hennessy; Patterson, 2011).

No contexto de aplicações sísmicas, uma das técnicas que consegue aproveitar bem o paralelismo das GPUs é a de inversão de forma de onda completa (*Full Waveform Inversion* - FWI), devido à natureza massivamente paralela dos cálculos envolvidos (Pandey *et al.*, 2022). Entretanto, conforme a escala dos problemas e o volume de dados cresce, a necessidade de se expandir ainda mais esse paralelismo e atender a demandas cada vez maiores por memória pede o uso de técnicas avançadas, como *checkpointing* (Symes, 2007) e compressão de dados

(Kukreja *et al.*, 2020), aliados a técnicas de decomposição de domínio para o uso de múltiplas GPUs (Fabien-Ouellet; Gloaguen; Girox, 2017).

1.1 OBJETIVOS

1.1.1 Objetivo Geral

Este trabalho tem como objetivo propor, implementar e validar estratégias eficientes para integrar técnicas de compressão de dados e *checkpointing* em uma aplicação de FWI de ondas acústicas em um ambiente multi-GPU, explorando as capacidades de paralelismo disponíveis em bibliotecas adequadas, incluindo OpenMP e outras ferramentas de suporte a programação paralela em GPUs.

1.1.2 Objetivos Específicos

Para alcançar esse objetivo geral, algumas metas específicas devem ser atingidas, como:

1. Analisar as técnicas existentes de *checkpointing* e compressão de dados em aplicações de computação de alto desempenho, identificando as abordagens mais adequadas para ambientes multi-GPU.
2. Estudar e validar técnicas de decomposição de domínio para escalabilidade eficiente em múltiplas GPUs, avaliando o uso de OpenMP e outras bibliotecas especializadas.
3. Implementar a estratégia mais eficiente encontrada, garantindo escalabilidade, facilidade de manutenção e melhoria no tempo de execução da aplicação.

2 REFERENCIAL TEÓRICO

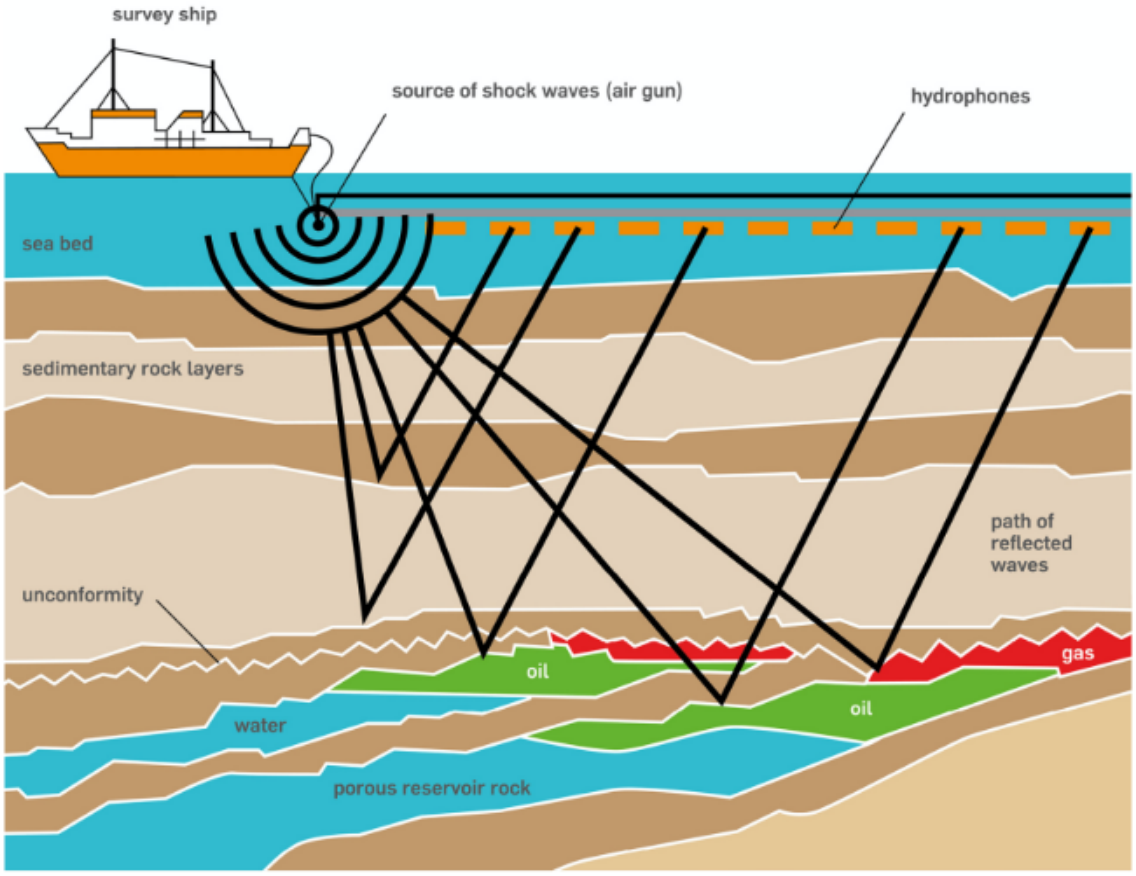
2.1 TÉCNICAS DE IMAGEAMENTO SÍSMICO

As técnicas de imageamento sísmico utilizam a simulação da propagação de ondas acústicas para criar imagens detalhadas de camadas subterrâneas, desempenhando um papel crucial na identificação e mapeamento de recursos como petróleo, gás, água e minerais (Virieux; Operto, 2009). Um exemplo notável de aplicação recente é a exploração de hidrocarbonetos na região do pré-sal da costa brasileira, conforme ilustrado na Figura 1.

O princípio fundamental dessas técnicas baseia-se na resolução de um problema inverso, no qual propriedades desconhecidas do meio são inferidas a partir de dados observacionais da propagação das ondas. O processo inicia-se com a construção de um modelo direto, utilizando equações diferenciais parciais (EDPs) para descrever a física envolvida. Os sinais coletados, gerados e refletidos nas camadas do subsolo, são comparados com os resultados obtidos por simulação numérica, e a discrepância entre ambos é gradualmente reduzida por meio de ajustes iterativos no modelo, até que se alcance uma correspondência satisfatória (Fichtner, 2011).

Embora seja muito eficaz para mapear as camadas subterrâneas, esse processo é extremamente custoso do ponto de vista computacional. Algoritmos baseados no método de diferenças finitas são amplamente utilizados para a solução numérica das EDPs em aplicações sísmicas. Entretanto, apesar da relativa simplicidade do método, a manutenção, a extensibilidade, a portabilidade e a adaptação desses programas apresentam grandes desafios.

Figura 1: Processo de coleta de dados para exploração geofísica



Fonte: Souza et at., 2022

2.2 FULL WAVEFORM INVERSION

A *Full Waveform Inversion* (FWI) é uma técnica de imageamento empregada para reconstruir modelos detalhados das propriedades físicas do subsolo. Seu princípio reside na formulação de um problema inverso, no qual parâmetros do meio, como a velocidade das ondas sísmicas e a densidade, são ajustados iterativamente para minimizar a diferença entre os dados sísmicos observados e os simulados (Fichtner, 2011). Para tanto, é necessário resolver repetidamente as equações diferenciais parciais que descrevem a propagação das ondas, tornando o FWI um processo altamente intensivo do ponto de vista computacional (Fabien-Ouellet; Gloaguen; Girox, 2017).

O método é amplamente utilizado na exploração de hidrocarbonetos, onde a precisão no mapeamento de reservatórios é fundamental para a redução de custos e

o aumento da eficiência. Além disso, a FWI mostra-se promissora em áreas emergentes, como o armazenamento de dióxido de carbono em cavernas geológicas, iniciativa alinhada à mitigação das mudanças climáticas.

Apesar dos benefícios, a aplicação da FWI enfrenta desafios significativos, principalmente devido ao elevado custo computacional, que cresce exponencialmente com a resolução e a complexidade do modelo. Isso se deve ao fato de que, a cada iteração, são necessárias simulações diretas e adjuntas das ondas sísmicas, exigindo grande capacidade de processamento e armazenamento de dados.

Nesse contexto, a computação de alto desempenho desempenha um papel fundamental. Tecnologias de HPC, como o uso de GPUs, aliadas a técnicas de otimização – incluindo *checkpointing* (que minimiza o retrabalho na reavaliação do modelo adjunto) e compressão de dados (que reduz o espaço em disco necessário e os custos de transferência entre host e dispositivos) – são empregadas para diminuir o tempo total do imageamento.

Dessa forma, o projeto que incorpora técnicas de decomposição de domínio pode ser estruturado em partes distintas, cada uma apresentando desafios específicos que devem ser considerados na distribuição do processamento em múltiplas GPUs.

2.2.1 Problema Direto

No cerne da *Full Waveform Inversion* está a simulação da propagação da onda acústica. Diversas equações que descrevem essa propagação existem, cada uma fundamentada em diferentes modelos físicos. Este trabalho utiliza como base a biblioteca Simwave, que adota um modelo acústico simplificado, considerando um meio isotrópico e com densidade constante (Souza et al., 2022). A equação diferencial de segunda ordem que descreve o deslocamento das partículas é dada por:

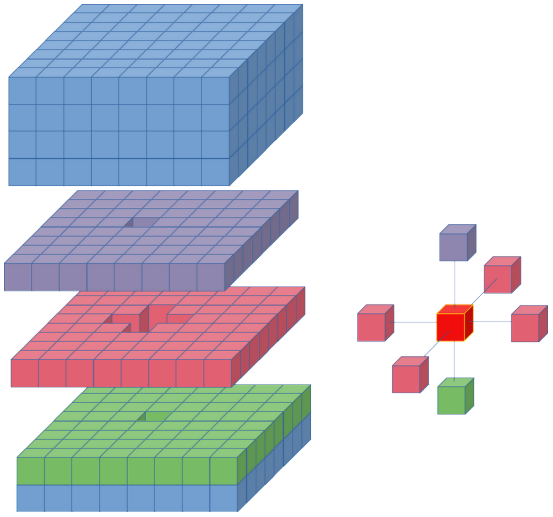
$$\frac{\partial^2 p}{\partial t^2}(x,t) - c^2(x)\nabla^2 p(x,t) = -\rho c^2(x)\nabla . b(x,t)$$

Nesta equação:

- p(x,t) representa a pressão escalar;
- c(x) é a velocidade da onda no meio;
- ρ é a densidade do meio, considerada constante;
- b(x,t) corresponde às forças externas do corpo;
- ∇^2 representa o operador laplaciano, onde $\nabla = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right)$, em coordenadas cartesianas.

Para viabilizar a solução numérica, essa equação deve ser discretizada. Embora diversos métodos possam ser empregados, o método de diferenças finitas com malha regular é amplamente utilizado na indústria devido à sua simplicidade e eficiência computacional (Souza et al., 2022). A discretização consiste na substituição das derivadas diferenciais contínuas por aproximações numéricas em uma grade de pontos, que pode ser bidimensional ou tridimensional, conforme a complexidade do problema. A malha espacial é definida por um espaçamento uniforme (por exemplo, $\Delta x, \Delta y, \Delta z$) em cada direção, enquanto o tempo é dividido em incrementos Δt.

Figura 2: Ilustração do padrão stencil



Fonte: Freire; Gomi; Senger, 2023

Algoritmo 1: Propagação de onda acústica

```
1: S = ComputeSubdomainBounds();   ▷ Compute begin/end for subdomains in z axis
2: for iteration ∈ [0, n] do
3:     for each subdomain s ∈ S do in parallel   ▷ Each device computes one subdomain in parallel
4:         for k ∈ [radius + s.begin_z, s.end_z − radius] do in parallel   ▷ Data parallelism occurs within each subdomain/device
5:             for j ∈ [stencil_radius, size_y − stencil_radius] do
6:                 for i ∈ [stencil_radius, size_x − stencil_radius] do
7:                     Compute p^{n+1}_{i,j,k} as described in Eq. 2
8:                 end for
9:             end for
10:        end for
11:        exchange_halos(d);
12:    end for
13: end for
```

Fonte: Freire; Gomi; Senger, 2023

A equação de onda bidimensional, discretizada utilizando diferenças centrais de segunda ordem tanto no tempo quanto no espaço, pode ser expressa na forma:

$$p_{i,j}^{n+1} = 2\,p_{i,j}^{n} - p_{i,j}^{n-1} + \frac{\Delta t^2 v^2}{\Delta x^2}\left(p_{i+1,j}^{n} - 2\,p_{i,j}^{n} + p_{i-1,j}^{n}\right) + \frac{\Delta t^2 v^2}{\Delta y^2}\left(p_{i,j+1}^{n} - 2\,p_{i,j}^{n} + p_{i,j-1}^{n}\right)$$

Nessa formulação, o valor da pressão em um ponto (i, j) no instante t é calculado a partir dos valores dos pontos vizinhos nas direções x e y, além dos valores na iteração atual e na iteração anterior. Essa dependência espacial é conhecida como padrão de stencil, o qual pode ser estendido ao caso tridimensional, conforme ilustrado na Figura 2.

A implementação computacional deste model é iterativa, com todos os pontos sendo atualizados a cada passo de tempo. O Algoritmo 1 exemplifica o cálculo dessa equação em um domínio 3D, conforme implementado pela biblioteca Simwave (Souza oet al., 2022).

2.2.2 Problema Adjunto

A segunda parte da solução da FWI consiste no chamado *problema adjunto*, que se baseia na minimização do erro entre a onda simulada e os sinais capturados no meio físico. Essa etapa é fundamental para ajustar iterativamente os parâmetros do modelo geológico, buscando uma correspondência satisfatória com os dados reais.

A minimização do erro entre a onda simulada e os sinais observados pode ser realizada por meio da quantificação de uma função de desajuste $J(m)$, que mede a

discrepância entre as formas de onda observadas $u^{obs}(xr,t)$ em receptores localizados em xr ao longo de um intervalo de tempo $T0$, e as formas de onda simuladas $u^{sim}(xr,t)$, computadas a partir de um modelo de superfície, que é o chamado problema direto (Fabien-Ouellet; Gloaguen; Girox, 2017).

Uma das técnicas de otimização mais utilizadas na solução de problemas adjuntos é o *checkpointing*, que consiste no armazenamento periódico de estados intermediários da simulação direta. O objetivo do *checkpointing* é minimizar a necessidade de recalcular a propagação da onda completa durante o problema adjunto, economizando tempo de execução e recursos computacionais (Symes, 2007).

No método utilizado pelo FWI, a resolução do problema adjunto exige a reconstrução de estados anteriores da propagação da onda para o cálculo dos gradientes e a atualização dos parâmetros do modelo. Entretanto, armazenar todos os estados a cada passo de tempo seria inviável devido ao alto consumo de memória. O *checkpointing* resolve esse problema salvando apenas alguns estados intermediários estrategicamente distribuídos. Durante a fase adjunta, os estados intermediários não armazenados são recomputados conforme necessário, utilizando os *checkpoints* como pontos de partida, reduzindo significativamente o custo computacional da recuperação.

O uso de checkpointing pode oferecer grandes vantagens ao executar o algoritmo de imageamento sísmico, pois reduz o consumo de memória ao custo de uma recomputação controlada, permitindo um melhor balanceamento entre uso de recursos e tempo de execução (Symes, 2007).

2.2.3 Compressão de dados

O emprego do checkpointing para otimizar a execução do FWI apresenta desafios relacionados ao uso de memória. A memória disponível nas GPUs muitas vezes não é suficiente para armazenar todos os dados necessários à simulação da propagação da onda (Kukreja et al., 2020). Dessa forma, o armazenamento periódico desses estados intermediários exige o uso da memória do *host* — que geralmente é mais abundante em um nó computacional — ou mesmo de dispositivos de armazenamento secundário, como SSDs e discos rígidos.

A transferência de dados entre GPUs e a memória primária do sistema ocorre por meio de barramentos PCIe, cuja velocidade máxima é limitada. Esse *overhead* de comunicação, causado pela largura de banda restrita do PCIe e pela latência da movimentação de dados, pode impactar significativamente o desempenho do algoritmo (Symes, 2007) e deve ser considerado ao definir a melhor estratégia de *checkpointing*. Além disso, mesmo a memória primária e o espaço em disco em sistemas de alto desempenho podem se revelar insuficientes, dependendo do número de *checkpoints* e do tamanho do grid computacional adotado.

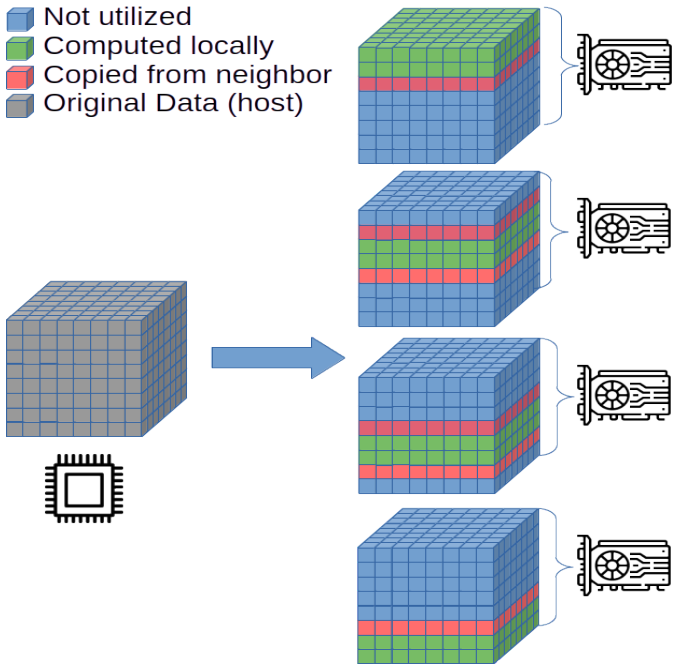
A compressão de dados surge como uma solução para esses desafios, reduzindo a quantidade de dados trafegados entre a GPU e o host, o espaço ocupado por cada *checkpoint* e, em alguns casos, acelerando a recuperação dos estados armazenados (Kukreja et al., 2020).

2.3 DECOMPOSIÇÃO DE DOMÍNIO

O uso de múltiplas GPUs para a resolução de um problema requer a chamada decomposição de domínio, uma técnica fundamental para a paralelização de algoritmos em computação científica. Essa abordagem permite dividir problemas complexos em subproblemas menores e gerenciáveis (Kirk; Hwu, 2016; Sanders; Kandrot, 2010). A decomposição de domínio é especialmente relevante em simulações de FWI, onde a alta demanda computacional exige estratégias eficientes de processamento paralelo, frequentemente suportadas por arquiteturas de GPUs.

Em geral, a decomposição de domínio consiste em subdividir o espaço computacional — por exemplo, a malha numérica que representa o subsolo em simulações geofísicas — em regiões menores chamadas subdomínios. Cada subdomínio pode ser processado de forma independente em diferentes GPUs, permitindo a execução simultânea dos cálculos e reduzindo o tempo total da simulação. No entanto, a subdivisão do domínio implica a necessidade de comunicação entre os subdomínios, pois os cálculos em uma GPU podem depender dos valores computados em GPUs vizinhas.

Figura 3: Representação visual de decomposição de domínio



Fonte: Elaborado pelo autor, 2025

Para garantir a consistência dos resultados, é essencial a implementação de mecanismos de comunicação e sincronização entre as unidades de processamento (NVIDIA, 2020). Em cálculos baseados em *stencil*, como os usados na discretização de equações diferenciais parciais, as bordas dos subdomínios contêm regiões de sobreposição chamadas de halos. Essas regiões precisam ser frequentemente atualizadas durante a execução do algoritmo, pois contêm informações necessárias para manter a continuidade da solução entre subdomínios. O custo da comunicação de halos pode se tornar um gargalo para o desempenho da decomposição de domínio, especialmente quando a comunicação ocorre com alta frequência ou em sistemas com alto custo de transferência de dados entre GPUs.

A Figura 3 ilustra um exemplo de decomposição de domínio, onde a malha computacional é dividida em várias fatias, cada uma processada por uma das quatro GPUs.

2.4 SIMWAVE

O Simwave é uma biblioteca Python que implementa um kernel de propagação de ondas acústicas e foi utilizada como base para a validação e implementação das técnicas de decomposição de domínio abordadas neste

trabalho. O projeto foi desenvolvido por pesquisadores da Universidade Federal de São Carlos (UFSCar) em colaboração com a Universidade de São Paulo (USP).

O Simwave possui um *front-end* desenvolvido em Python, para "fornecer uma interface amigável que simplifica o desenvolvimento de aplicações e facilita a integração com outras bibliotecas científicas, como o SciPy, entre outras" (Souza *et at.,* 2022, p. 13, tradução nossa), enquanto que

> Todas as estratégias de processamento paralelo e otimizações específicas de *hardware* são implementadas no *back-end*. Os componentes críticos de desempenho são desenvolvidos no *back-end*, que é escrito em ANSI C (sequencial) ou em C com suporte para paralelismo (por exemplo, OpenMP, OpenACC, etc.). (Souza *et at.,* 2022, p. 13, tradução nossa).

O uso de C no *back-end* possibilitou a experimentação de diversas técnicas e ferramentas de paralelismo, como *multi-threading*, *offloading* em GPUs e bibliotecas como CUDA, OpenACC e OpenMP, viabilizando otimizações para diferentes plataformas.

Além de fornecer um kernel para propagação de ondas acústicas, o Simwave também serve como base para a implementação de um FWI completo, incluindo técnicas de *checkpointing* e compressão de dados em GPU. Este trabalho foca na implementação da decomposição de domínio no Simwave, utilizando OpenMP com offloading para GPU devido ao desempenho superior apresentado em relação às alternativas testadas, conforme demonstrado na Tabela 1.

Tabela 1: Tempo de execução (segundos) e speedup da simulação 3D da propagação de onda

Hardware	Back-end	Compiler	SO=2		SO=4		SO=8	
			Time	S	Time	S	Time	S
6148 - 1 core	C	gcc	1642.41	1.0	2565.88	1.0	3909.55	1.0
6148 - 2 cores	OpenMP	gcc	901.51	1.8	1360.49	1.9	2048.54	1.9
6148 - 4 cores	OpenMP	gcc	475.52	3.5	716.31	3.6	1081.00	3.6
6148 - 8 cores	OpenMP	gcc	248.84	6.6	374.98	6.8	569.71	6.9
6148 - 20 cores	OpenMP	gcc	186.98	8.8	272.56	9.4	429.54	9.1
6148 - 40 cores	OpenMP	gcc	110.72	14.8	171.11	15.0	347.09	11.3
RTX 2080 Super	OpenMP	clang	72.46	22.7	93.95	27.3	130.23	30.0
RTX 2080 Super	OpenACC	pgcc	48.02	34.2	67.68	37.9	103.95	37.6
V100	OpenMP	clang	**28.30**	**58.0**	**40.36**	**63.6**	**63.12**	**61.9**
V100	OpenACC	pgcc	37.13	44.2	50.10	51.2	68.13	57.4

Fonte: Souza et at., 2022

2.5 TRABALHO ANTERIOR

O presente trabalho dá continuidade a um projeto de Iniciação Científica do autor, intitulado "Simulação da onda acústica para FWI de alta frequência em múltiplas GPUs", no qual foram estudadas diversas estratégias para a paralelização de um kernel de propagação de ondas em múltiplas GPUs. Nesse contexto, foram implementadas duas estratégias de mapeamento explícito de dados, utilizando funções da biblioteca OpenMP e funções nativas CUDA, além de duas estratégias de mapeamento implícito — através das diretivas *omp target data map* e do uso de *Unified Virtual Addressing* (UVA) da biblioteca de *runtime* CUDA.

Tabela 2: Tempo de execução (segundos) e speedup para a execução da propagação da onda em 4 GPUs

			Baseline	OMP		UVA		Auxiliary		Coupled		Auxiliary		Coupled	
	Grid	# of	(1 GPU)	Data map				OMP		OMP		CUDA		CUDA	
SO	size	iters.	Time	Time	S	Time	S	Time	S	Time	S	Time	S	Time	S
2	256³	400	0,59	0,9	0,66	1,05	0,56	0,68	0,87	0,74	0,80	0,62	0,95	0,6	0,98
		4000	4,2	4,31	0,97	4,66	0,90	2,4	1,75	2,36	1,78	1,72	2,44	1,65	2,55
	512³	400	3,62	2,87	1,26	3,05	1,19	1,79	2,02	1,68	2,15	1,54	2,35	1,6	2,26
		4000	31,44	17,45	1,80	15,01	2,09	9,9	3,18	9,9	3,18	8,71	3,61	8,58	3,66
	1024³	400	30,17	15,23	1,98	17,08	1,77	10,16	2,97	10,12	2,98	9,67	3,12	9,9	3,05
		4000	269,91	96,54	2,80	92,99	2,90	73,49	3,67	73,14	3,69	69,82	3,87	70,85	3,81
4	256³	400	0,63	1,07	0,59	1,21	0,52	0,75	0,84	0,67	0,94	0,69	0,91	0,63	1,00
		4000	4,76	6,31	0,75	5,58	0,85	2,89	1,65	2,52	1,89	1,83	2,60	1,77	2,69
	512³	400	4,78	3,65	1,31	3,52	1,36	2,01	2,38	2,05	2,33	1,78	2,69	1,82	2,63
		4000	42,99	24,65	1,74	20,73	2,07	13,75	3,13	13,48	3,19	11,44	3,76	11,81	3,64
	1024³	400	36,82	19,74	1,87	22,13	1,66	11,86	3,10	11,43	3,22	11,41	3,23	11,18	3,29
		4000	337,47	135,6	2,49	144,67	2,33	91,77	3,68	91,2	3,70	87,41	3,86	88,62	3,81
8	256³	400	0,78	1,51	0,52	1,48	0,53	0,85	0,92	0,86	0,91	0,68	1,15	0,75	1,04
		4000	6,36	10,89	0,58	8,31	0,77	3,39	1,88	3,02	2,11	2,05	3,10	2,15	2,96
	512³	400	6,54	5,01	1,31	4,61	1,42	2,46	2,66	2,48	2,64	2,35	2,78	2,34	2,79
		4000	60,56	37,07	1,63	31,93	1,90	18,65	3,25	18,31	3,31	16,04	3,78	16,43	3,69
	1024³	400	51,58	26,62	1,94	33,77	1,53	15,94	3,24	15,91	3,24	15,39	3,35	15,09	3,42
		4000	484,92	212,68	2,28	261,45	1,85	132,17	3,67	130,87	3,71	126,01	3,85	127,8	3,79

Fonte: Freire; Gomi; Senger, 2023

Através desse trabalho, foi possível aferir, tanto qualitativa quanto quantitativamente, as vantagens e desvantagens de cada estratégia. Entre os resultados obtidos, destacamos que:

> Os métodos implícitos testados (OpenMP *data map* e CUDA *Unified Virtual Addressing*) apresentaram resultados pouco expressivos em termos de tempo de execução, alcançando um *speedup* de pouco mais de 2x apenas para um grande número de iterações em *grids* de maior tamanho. Embora o *Unified Virtual Addressing* possa ser recomendado para aumentar a

> capacidade de memória e obter pequenos ganhos de desempenho, o *target data map* do OpenMP se mostrou insuficiente para implementações multi-GPU de algoritmos do tipo *stencil* e, portanto, não é recomendado. (Freire; Gomi; Senger, 2023, p. 10-11, tradução nossa).

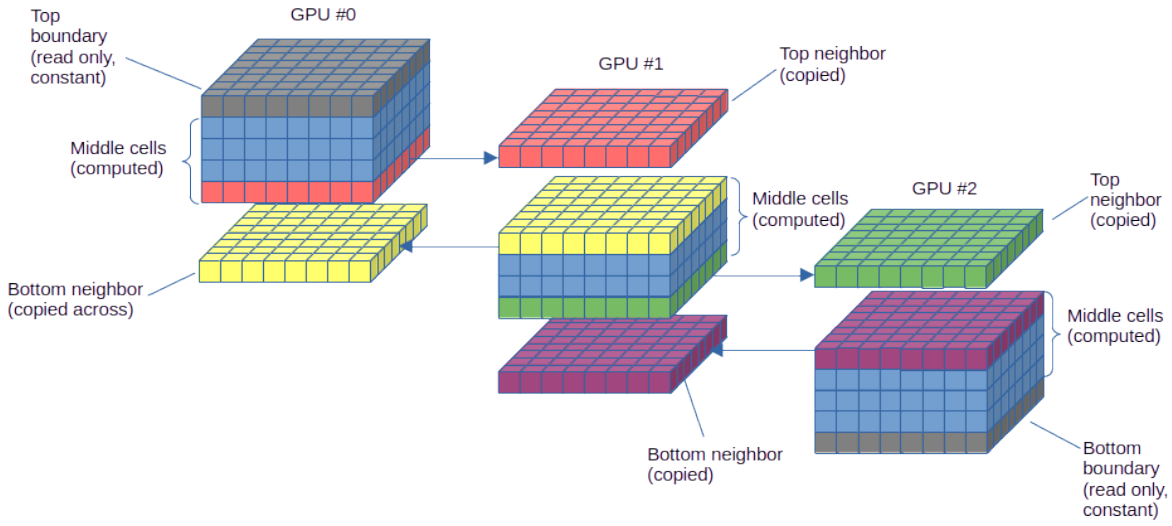
Esses resultados nos levaram a desconsiderar essas técnicas em nossa abordagem atual. Em relação às bibliotecas utilizadas, observamos que os **ganhos de desempenho ao utilizar as funções nativas da biblioteca de runtime CUDA** não foram significativos o suficiente para justificar sua inclusão no código, conforme evidenciado na Tabela 2.

De forma subjetiva, também foi possível comparar a dificuldade de implementação e manutenção das estratégias, pelos autores intituladas como "*Auxiliary Halos*" (Figura 4) e "*Coupled Halos*" (Figura 5). Concluiu-se que:

> Ao comparar ambas as estratégias explícitas, observa-se que, embora o uso de *arrays* auxiliares para o *halo* possibilite um melhor sobreposição entre a cópia de dados e a computação, os benefícios dessa abordagem se mostraram inconsistentes, além de exigir uma implementação mais complexa e suscetível a erros. Por outro lado, a utilização de um único array proporciona melhor legibilidade do código, mantendo um desempenho similar. (Freire; Gomi; Senger, 2023, p.11, tradução nossa)

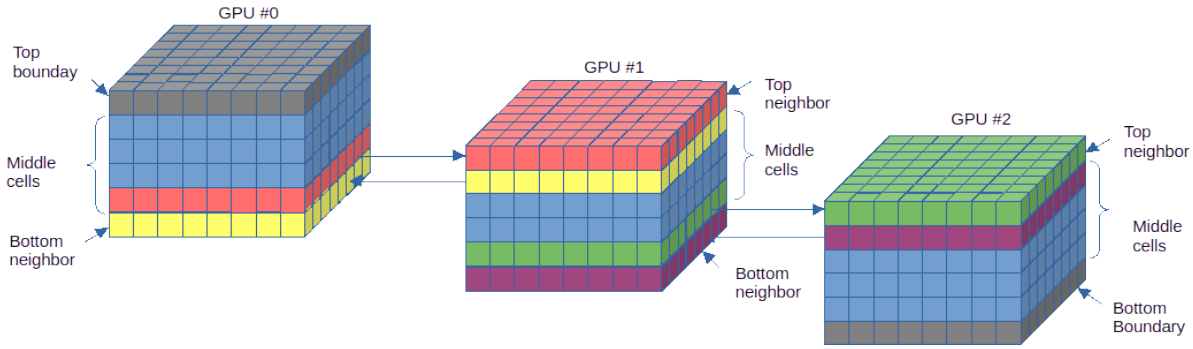
Tendo em vista esses resultados, o presente trabalho utiliza a estratégia de "*Coupled Halos*", utilizando as funções nativas da biblioteca OpenMP para realizar a transferência de dados do Halo.

Figura 4: Estratégia de decomposição de domínio intitulada de "*Auxiliary Halos*"



Fonte: Freire; Gomi; Senger, 2023

Figura 5: Estratégia de decomposição de domínio intitulada de "*Coupled Halos*"



Fonte: Freire; Gomi; Senger, 2023

3 METODOLOGIA

A pesquisa tem caráter exploratório e quantitativo, utilizando o método indutivo para a obtenção de resultados (Lakatos; Marconi, 2003). O trabalho foi estruturado nas seguintes etapas:

1. Levantamento bibliográfico sobre o estado da arte, teoria e implementação de algoritmos de imageamento sísmico, com foco no FWI. A pesquisa foi realizada por meio de expressões de busca relacionadas ao tema, utilizando o portal de periódicos da CAPES e materiais indicados pelo orientador.
2. Levantamento bibliográfico sobre as técnicas de *checkpointing*, compressão de dados e decomposição de domínio em algoritmos de computação de alto desempenho para GPUs. As fontes foram obtidas no portal de periódicos CAPES e por meio de indicações do orientador.
3. Estudo da teoria e implementação da biblioteca Simwave (Souza et al., 2022), com o objetivo de entender a implementação de um FWI a partir de sua base de código.
4. Estudo das bibliotecas e técnicas utilizadas na implementação de decomposição de domínio, buscando um melhor aprofundamento nos métodos adotados.
5. Implementação de uma versão final que combina as técnicas mencionadas, com o objetivo de garantir os objetivos específicos descritos nas etapas anteriores.
6. Análise dos resultados obtidos.

4 DESENVOLVIMENTO

4.1 CONSIDERAÇÕES INICIAIS

O presente trabalho baseia-se em conhecimentos previamente adquiridos durante um trabalho de iniciação científica (Freire; Gomi; Senger, 2023) e dá continuidade ao desenvolvimento realizado pelo grupo de pesquisa do professor Hermes Senger. A biblioteca Simwave, juntamente com outros projetos derivados dela, como o desenvolvimento de um FWI, possibilitou a aplicação e o estudo de diversas técnicas de computação de alto desempenho, como *checkpointing* e compressão de dados, em um contexto prático e realista.

O desenvolvimento deste trabalho ocorreu em três etapas principais. Primeiramente, foi necessário familiarizar-se com a base de código existente, desenvolvida pelo grupo de pesquisa da universidade. A implementação existente utiliza o Simwave como propagador de ondas e implementa um FWI completo, utilizando a biblioteca OpenMP para processamento em GPUs, incluindo técnicas de *checkpointing* e compressão de dados em GPUs por meio da biblioteca nvcomp.

Na segunda etapa, foram estudadas as técnicas de *checkpointing* e compressão de dados, dado que a compreensão plena dessas técnicas é essencial para o desenvolvimento do trabalho. Esse estudo incluiu tanto a literatura disponível quanto a inspeção detalhada do código previamente desenvolvido. Finalmente, foi realizada a implementação das técnicas de multi-GPU, que fazem parte da solução final do trabalho. Este trabalho dá continuidade ao desenvolvimento de um FWI, liderado pelo grupo de pesquisa do Simwave, sob a orientação do professor Hermes Senger. A implementação utiliza OpenMP com *target offloading* para execução em GPUs e emprega o Simwave (Souza et al., 2022) como propagador de ondas acústicas em um meio isotrópico com densidade constante.

O processo segue um ciclo iterativo, no qual, a cada iteração, a onda simulada é propagada a partir de um modelo de velocidade inicial. Em seguida, calcula-se o erro em relação aos dados observados, permitindo a obtenção do campo de onda adjunto. Esse campo é usado para computar o gradiente da função objetivo, que, por sua vez, serve para atualizar o modelo de velocidade na próxima

iteração. O ciclo se repete até que a solução convirja, refinando progressivamente o modelo.

Para otimizar a eficiência computacional, foram adotadas duas técnicas principais. A primeira é o *checkpointing*, que reduz a demanda de memória ao armazenar apenas estados estratégicos da propagação das ondas, recomputando

Algoritmo 2: Implementação base de um FWI

for `iteration ∈ [0, max_iterations]` **do**
    // Forward modeling: Solve the wave equation
    for `t ∈ [0, total_time]` **do**
        Compute synthetic wavefield $u_{sint}(x, z, t)$
    end for
    // Compute the data residual
    Compute $erro(xr, t) = d_{obs}(x_r, t) - d_{sint}(x_r, t)$
    // Adjoint wavefield computation
    for t ∈ [total_time, 0] **do**
        Compute adjoint wavefield $u_{adj}(x, z, t)$
    end for
    // Compute the gradient of the objective function
    for each (x, z) in the model domain **do**
        Compute gradient $g(x, z)$ from correlation of forward and adjoint wavefields
    end for
    // Update the velocity model
    for each (x, z) in the model domain **do**
        Update velocity $v(x, z) = v(x, z) + \alpha \cdot g(x, z)$
    end for
end for

Fonte: Elaborado pelo autor

os estados intermediários quando necessário. Isso torna viáveis simulações de grande porte sem comprometer o desempenho. A segunda técnica é a compressão de dados em GPU, que diminui o volume de informações transferidas entre a CPU e a GPU, minimizando a sobrecarga de comunicação e otimizando a utilização da memória disponível.

Com essas estratégias, a implementação combina precisão e eficiência, aproveitando paralelismo via OpenMP offloading e técnicas de otimização para lidar com a alta complexidade computacional do FWI. A lógica geral do algoritmo está descrita no Algoritmo 2, que apresenta a sequência de operações utilizadas no propagador de ondas e no cálculo do gradiente.

4.2 IMPLEMENTAÇÃO

A implementação realizada dá continuidade a trabalho desenvolvido anteriormente pelo grupo de pesquisa do Simwave, liderado pelo professor Hermes Senger. Em trabalhos anteriores, foram desenvolvidas as técnicas de *checkpointing*, que posteriormente foram aprimoradas com o uso de compressão de dados em GPU com o intuito de melhorar o desempenho da execução e diminuir a quantidade de memória necessária para a execução de um FWI completo.

Este trabalho também é a aplicação de estudo realizado anteriormente pelo autor (Freire; Gomi; Senger, 2023) durante um projeto de iniciação científica, no qual foram investigadas quais as melhores estratégias e quais ferramentas utilizar para a decomposição de domínio no projeto Simwave.

A estratégia de decomposição de domínio exercida utiliza as funções *cudaMalloc* e *cudaMemcpy* no lugar das diretivas *target data map* utilizadas na implementação original. Dessa forma, a alocação e transferência de dados entre host e dispositivos é feita de maneira explícita. Esse controle maior nos permite alocar e transferir somente os dados necessários, reduzindo o tempo utilizado para transferência de dados, e aproveitando a maior quantidade de memória disponível ao se combinar múltiplos dispositivos.

4.2.1 Divisão do problema

A solução do FWI se dá a partir do cálculo do gradiente entre uma onda simulada e a onda obtida em campo. O cálculo do gradiente é realizado através da função *gradient()*, que executa a simulação da propagação de onda e realiza o salvamento dos checkpoints. Na implementação original, todos os dados de um checkpoint são comprimidos através da biblioteca *nvcomp* antes de serem transferidos para a memória do *host*; similarmente, os *checkpoints* são transferidos ainda comprimidos para os dispositivos, onde são descomprimidos antes de serem utilizados na computação.

A implementação original divide o cálculo do gradiente em três funções distintas: a execução da propagação da onda com o salvamento de *snapshots*; a execução da propagação da onda a partir de *snapshots* previamente salvas; e o

cálculo do gradiente; sendo esta responsável por iniciar a execução das demais. É nesta última função também que os dados são alocados e copiados para o dispositivo que realizará o cálculo do gradiente do campo.

4.2.2 Decomposição de domínio

Algoritmo 3: FWI implementado incluindo com decomposição de domínio

```
1: for iteration ∈ [0, max_iterations] do
2:       // Forward modeling: Solve the wave equation
3:       for each device d do
4:             Distribute domain slices to device d, including halo regions
5:       end for
6:       for t ∈ [0, total_time] do
7:             Compute synthetic wavefield on each device
8:             Exchange halo regions between neighboring devices
9:       end for
10:       // Compute the data residual
11:       Compute residual
12:       // Adjoint wavefield computation
13:       for t ∈ [total_time, 0] do
14:             Compute adjoint wavefield on each device
15:             Exchange halo regions between neighboring devices
16:       end for
17:       // Compute the gradient of the objective function
18:       for each in the model domain do
19:             Compute gradient from correlation of forward and adjoint wavefields
20:       end for
21:       // Update the velocity model
22:       for each in the model domain do
23:             Update velocity
24:       end for
25: end for
```

Fonte: Elaborado pelo autor

Seguindo os conhecimentos adquiridos em trabalho anterior, a estratégia escolhida envolve subdividir o grid tridimensional de pontos em "fatias" de tamanho igual, cada qual sendo executada em um dispositivo, como exemplificado na Figura 3. Esses segmentos criados devem incorporar também uma cópia de cada região de *halo* que é calculada por um dispositivo "vizinho", que deve ser atualizada mutuamente após cada iteração dos métodos implementados.

O Algoritmo 3 descreve as etapas realizadas pelo algoritmo incluindo as etapas de transferência de dados entre dispositivos. Note que o processo de troca de *halo* é executado várias vezes durante a execução de um FWI completo, sendo necessária após cada *timestep*. Assim sendo, tecnologias de comunicação entre

dispositivos, tais quais NVLink (NVidia, 2025), e comunicação direta via PCIe são essenciais para que se atinja um bom desempenho utilizando essas abordagens.

4.2.3 Implementação

A execução de código em GPUs utilizando OpenMP é possível graças à diretiva *target*, que cria uma seção de código que deverá ser executada em um dispositivo externo. O desenvolvimento do processamento em GPUs na biblioteca OpenMP é relativamente recente, mas é capaz de combinar bons resultados com baixa complexidade de implementação.

A implementação completa realizada neste trabalho pode ser encontrada em repositório público (Nicolau, 2025), não contido completo no presente trabalho por conta de sua extensão.

A estratégia adotada envolve a criação de uma região paralela para cada *device* disponível através da cláusula *num_threads*, conforme demonstrado no Código 1. Cada região é executada em uma *thread* de CPU separada, e suas numerações são sequenciais e iniciadas em 0. Como a numeração das regiões paralelas segue o mesmo padrão que os *devices* disponíveis, a região paralela fica responsável pela GPU de mesmo número, cujo identificador fica armazenado na variável *device_id*, que também identifica o *device* da região para as instruções da biblioteca CUDA utilizadas posteriormente.

Código 1: Criação das regiões paralelas

```
608     int host = omp_get_initial_device();
609     #pragma omp parallel num_threads(NUM_DEVICES) shared(us)
610     {
611         // get default device
612         int device_id = omp_get_thread_num();
613         cudaSetDevice(device_id);
```

Fonte: Elaborado pelo autor

Outra ferramenta importante é o uso da cláusula *is_device_ptr* nas diretivas *omp target*. Essa diretiva instrui à biblioteca OpenMP que o ponteiro utilizado já foi alocado em, e se refere a, uma região de memória da GPU em uso. Através da cláusula *device*, cujo uso está demonstrado no Código 2, é possível especificar em qual dispositivo o *kernel* será executado.

Código 2: Uso das cláusulas *is_device_ptr* e *device*

```
404         int i_ini = stencil_radius;
405         int i_fim = individual_nz+stencil_radius;
406
407         if (device_id == 0)
408             i_ini = stencil_radius + stencil_radius;
409         if(device_id == NUM_DEVICES-1)
410             i_fim = individual_nz;
411
412 #pragma omp target teams distribute parallel for collapse(3) device(device_id) \
413     is_device_ptr(u, snapshot_d_prev, snapshot_d_current, d_velocity, d_damp,  \
414                         d_wavelet, d_coeff, src_points_interval,               \
415                         d_src_points_values, src_points_values_offset, top_u,  \
416                         bottom_u)
417         for(size_t i = i_ini; i < i_fim; i++) {
418             for(size_t j = stencil_radius; j < nx - stencil_radius; j++) {
419                 for(size_t k = stencil_radius; k < ny - stencil_radius; k++) {
420                     // index of the current point in the grid
421                     size_t domain_offset = (i * nx + j) * ny + k;
```

Fonte: Elaborado pelo autor

A combinação das duas ferramentas listadas permite um controle manual da execução de códigos otimizados através de OpenMP. Esse controle nos permite aproveitar o melhor desempenho oferecido pelas funções de gerenciamento de memória da biblioteca de runtime CUDA, quando comparadas às funções do OpenMP (Freire; Gomi; Senger, 2023). Através dessa são realizadas as alocações, inicializações e atualizações dos dados nos dispositivos.

Tendo em vista a mudança na dimensão dos vetores sobre os quais cada *loop* deverá trabalhar, o cálculo do novo tamanho e posições iniciais e finais de cada estrutura de repetição é recalculado a depender do dispositivo. O valor da posição no eixo Z, no laço mais externo da estutura, deve abranger pontos distintos para as GPUs responsáveis pelo topo e pela base do grid, e são computados antes do início da execução (Código 2).

Similarmente, a atualização dos valores do *halo* dependem da existência de um vizinho, e, portanto, são tratadas a depender do índice do dispositivo, conforme demonstrado no Código 3. A atualização ocorre através da função *cudaMemcpyPeer*, que força a comunicação direta entre GPUs quando disponível no sistema. Para que os valores não sejam alterados antes de serem utilizados em seus respectivos cálculos, foram utilizadas barreiras de sincronização, que só permitem que a execução adiante continue quando todas as *threads* de execução cheguem naquele ponto.

Código 3: Atualização da região do *halo* após cada iteração

```
462         #pragma omp barrier
463         if (device_id > 0) {
464             cudaMemcpyPeer(top_u + (individual_nz + stencil_radius) * nx * ny * sizeof(f_type), //Dst
465                     device_id-1,                                                                //Dst device
466                     u + stencil_radius * nx * ny * sizeof(f_type),                              //Src
467                     device_id,                                                                  //Src device
468                     stencil_radius * nx * ny * sizeof(f_type));                                 //Size
469         }
470
471         if (device_id < NUM_DEVICES - 1) {
472             cudaMemcpyPeer(bottom_u,                          //Dst
473                     device_id+1,                                 //Dst device
474                     u + individual_nz * nx * ny * sizeof(f_type),   //Src
475                     device_id,                                   //Src device
476                     stencil_radius * nx * ny * sizeof(f_type)); //Size
477         }
478
```

Fonte: Elaborado pelo autor

5 RESULTADOS

Os resultados descritos foram obtidos em um nó computacional dotado de 4 GPUs NVidia V100 conectadas via NVLink. Cada dispositivo possui 5120 núcleos CUDA e 32 GB de memória HBM2.

Os testes avaliam o tempo de execução para cada diferente implementação, sendo elas:

1. *Baseline*: Implementação single-gpu, que incorpora *checkpointing* mas não utiliza compressão de dados
2. *Compression*: Implementação single-gpu que utiliza *checkpointing* e compressão de dados
3. *Multi*: Implementação multi-gpu, que utiliza *checkpointing* mas não incorpora compressão de dados
4. *Multi compression*: Implementação multi-gpu que incorpora *checkpointing* e compressão de dados.

Os testes executados realizam o cálculo do gradiente com valores *float* de 32 bits, em um grid de tamanho variável e utilizam 5 *checkpoints*. Os valores de Space Order, número de *timesteps* e tamanho do domínio foram variados a fim de verificarmos a influência que os mesmos exercem no tempo de execução do cálculo do gradiente.

A Tabela 3 contém os valores de tempo de execução e *speedup* de cada implementação para cada combinação de *space order*, *grid size* e número de *timesteps*. A tabela foi colorida de forma a refletir o ganho de desempenho observado.

Tabela 3: Tempo de execução e speedup de todas as implementações

Space Order	Grid Size	# of timesteps	Baseline	Compression		Multi		Multi compression	
			Time (s)	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
2	50^3	500	6,22	6,06	1,03	13,07	0,48	13,28	0,47
		1000	19,81	18,79	1,05	49,36	0,40	49,65	0,40
	100^3	500	21,05	21,12	1,00	16,40	1,28	16,44	1,28
		1000	75,73	75,82	1,00	59,88	1,26	60,61	1,25
	200^3	500	124,68	124,28	1,00	43,59	2,86	43,87	2,84
		1000	476,98	476,92	1,00	161,02	2,96	162,58	2,93
	300^3	500	396,01	394,16	1,00	113,68	3,48	111,59	3,55
		1000	1526,33	1523,71	1,00	427,65	3,57	426,78	3,58
4	50^3	500	6,28	6,17	1,02	13,23	0,47	13,88	0,45
		1000	20,31	19,79	1,03	49,87	0,41	50,28	0,40
	100^3	500	22,95	22,34	1,03	17,80	1,29	17,80	1,29
		1000	83,45	82,69	1,01	65,22	1,28	65,04	1,28
	200^3	500	140,27	139,71	1,00	48,70	2,88	48,61	2,89
		1000	538,95	539,07	1,00	181,00	2,98	181,67	2,97
	300^3	500	456,34	454,20	1,00	129,69	3,52	127,76	3,57
		1000	1765,07	1762,10	1,00	489,19	3,61	488,43	3,61
8	50^3	500	7,02	6,87	1,02	13,45	0,52	13,42	0,52
		1000	21,59	21,27	1,01	50,58	0,43	50,09	0,43
	100^3	500	28,05	28,05	1,00	19,62	1,43	19,22	1,46
		1000	104,08	103,52	1,01	70,31	1,48	70,67	1,47
	200^3	500	168,71	168,14	1,00	57,81	2,92	57,72	2,92
		1000	652,43	651,62	1,00	217,27	3,00	217,49	3,00
	300^3	500	665,59	661,02	1,01	176,80	3,76	175,18	3,80
		1000	2589,75	2577,21	1,00	676,66	3,83	675,61	3,83

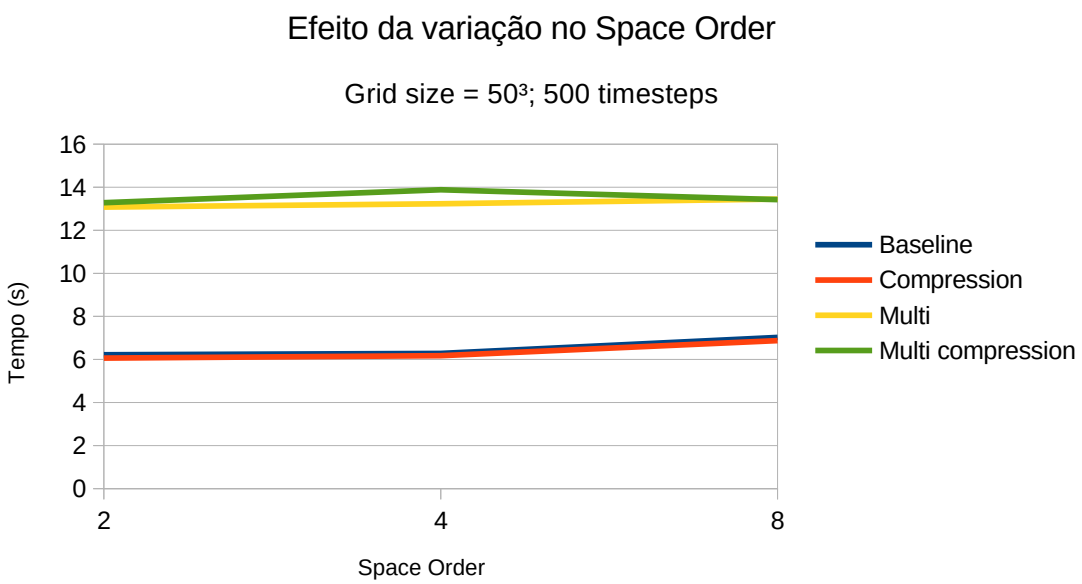
Fonte: Elaborado pelo autor

5.1 EFEITO DA VARIAÇÃO DO SPACE ORDER

Através dos resultados obtidos e dispostos abaixo, pudemos observar que o tamanho da *stencil,* definido como metade do valor do *Space Order*, não influencia consideravelmente no *speedup* de nenhuma das implementações. O efeito do aumento no *Space Order* pode ser observado no Gráfico 2 e no Gráfico 3, onde o aumento do raio da *stencil* contribui bastante para o aumento do tempo de execução para as soluções *single-gpu*, principalmente ao aumentarmos o tamanho do grid e o número de *timesteps*; mas não afetam significantemente o tempo de execução para as implementações multi-gpu.

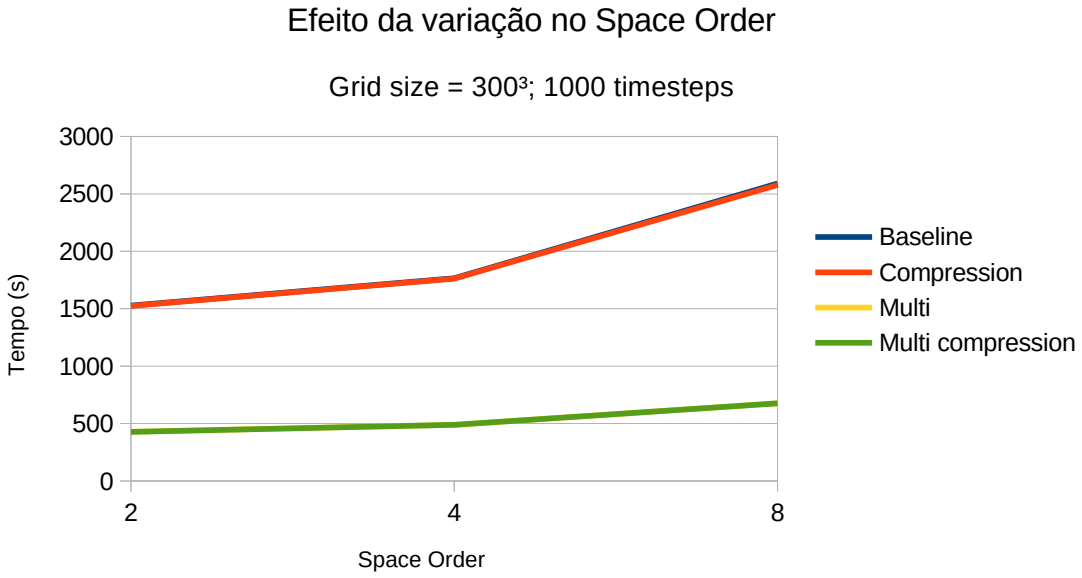
Isso pode ser explicado pela relativa baixa quantidade de dados trocados entre gpus ao se limitar a atualização apenas aos dados necessários, aliado ao uso da tecnologia NVLink, que permite que todos os dispositivos se comuniquem diretamente, sem a intervenção da CPU. Devido ao baixo custo de transferência, o processamento mais complexo afeta mais o tempo de execução em *single-gpu* do que o tempo de atualização dos *halos*.

Gráfico 1: Efeito da variação do Space Order no tempo de execução



Fonte: Elaborado pelo autor

Gráfico 2: Efeito da variação do Space Order no tempo de execução

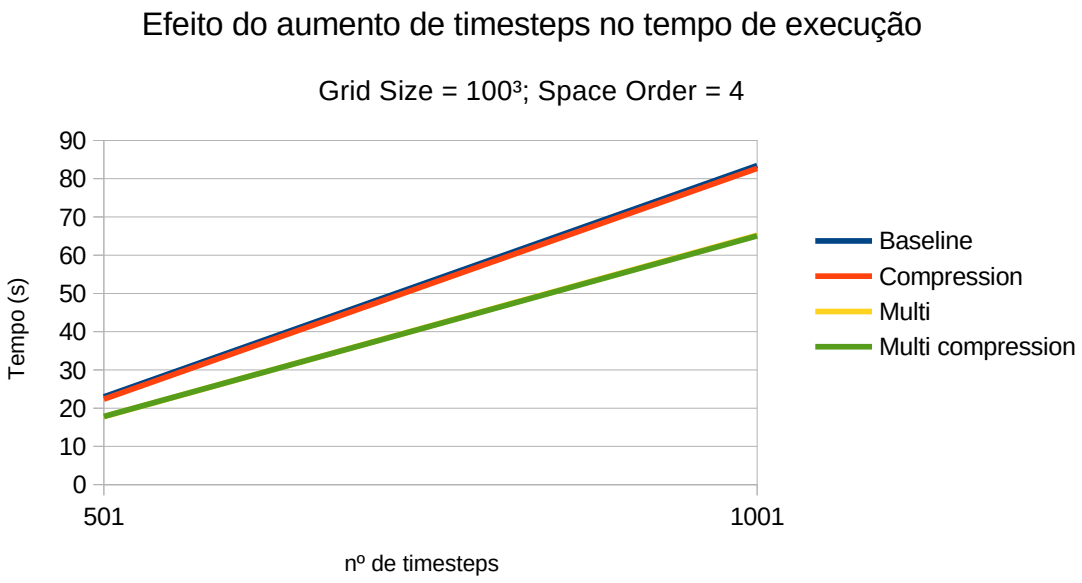


Fonte: Elaborado pelo autor

5.2 EFEITO DA VARIAÇÃO DO NÚMERO DE TIMESTEPS

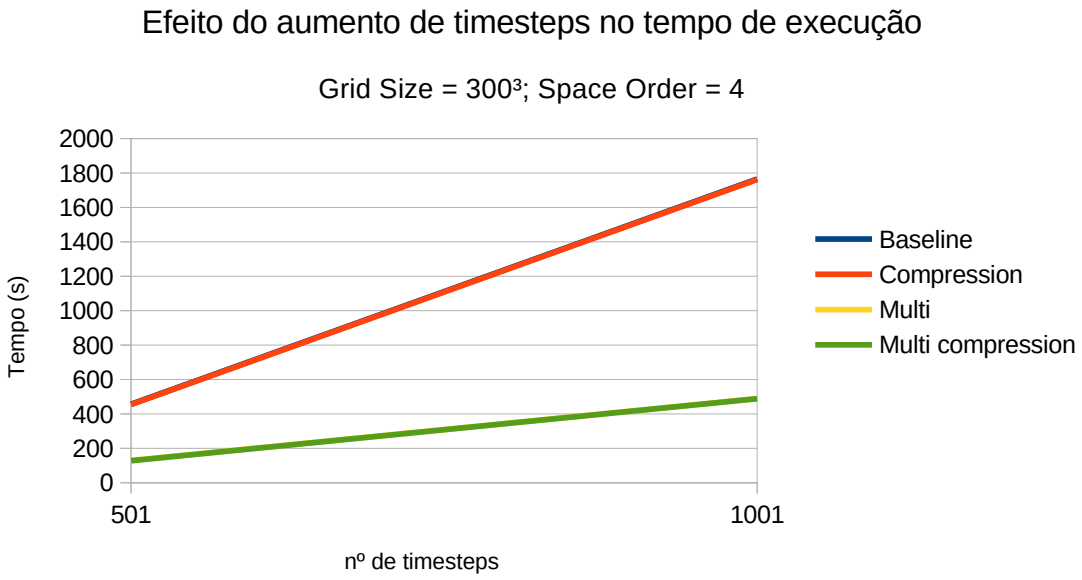
Como podemos observar no Gráfico 4, o aumento no número de *timesteps* não afeta na mesma proporção o tempo de execução das implementações *multi-gpu* em comparação às implementações *single-gpu* em grids de dimensões acima de 100^3, conforme visto no Gráfico 5. Isso indica escalabilidade dessa solução para a execução de problemas maiores, e demonstra novamente que o custo computacional da troca do halo é inferior aos ganhos obtidos no cálculo em múltiplos dispositivos.

Gráfico 3: Efeito da variação no número de timesteps



Fonte: Elaborado pelo autor

Gráfico 4: Efeito da variação no número de timesteps

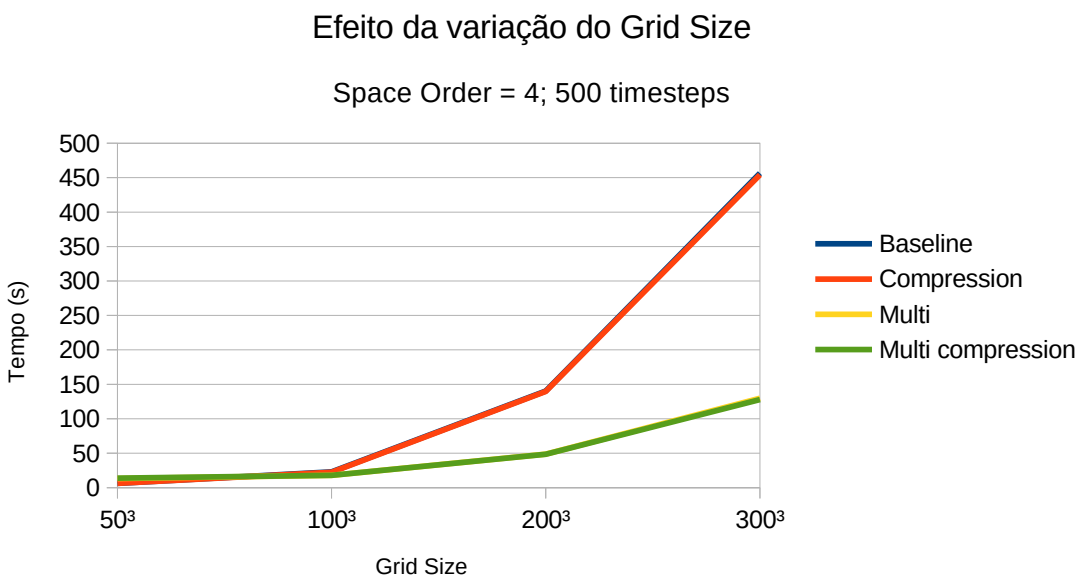


Fonte: Elaborado pelo autor

5.3 EFEITO DA VARIAÇÃO DO TAMANHO DE DOMÍNIO

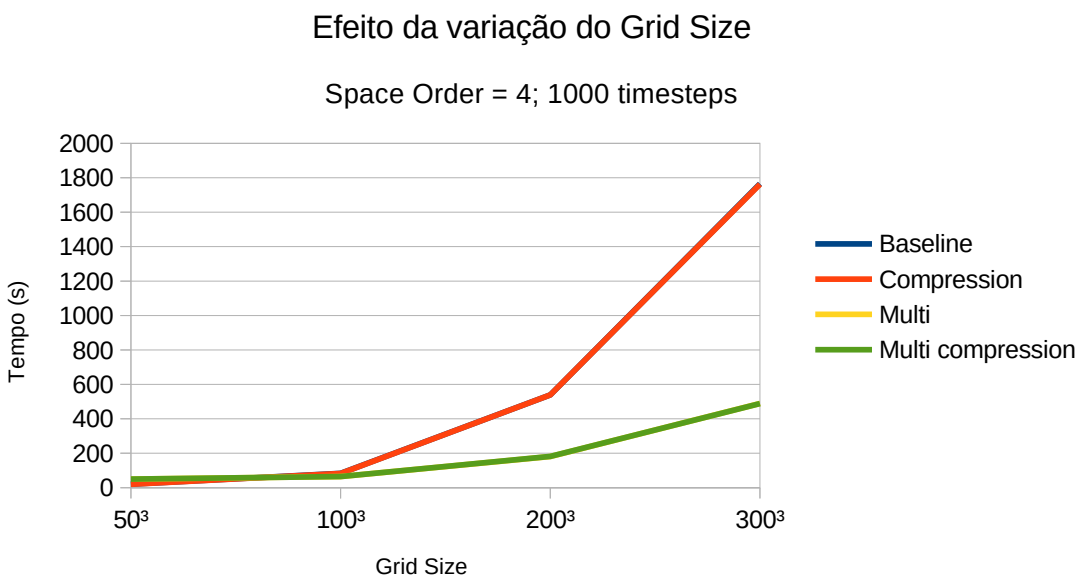
Através dos gráficos 5 e 6 e da Tabela 3, podemos constatar que o fator que mais influencia o *speedup* no uso da abordagem multi-gpu adotada é o tamanho do domínio. Vale observar que para problemas de tamanho pequeno, há perda de desempenho, o que pode ser atribuído a dois fatores importantes: o custo fixo da alocação, inicialização e atualização dos valores entre o *host* e as gpus; e ao aumento do custo relativo da atualização do *halo*, que nesses casos ultrapassam o ganho de desempenho ao se usar *multi-gpus*.

Gráfico 5: Efeito da variação no tamanho do grid



Fonte: Elaborado pelo autor

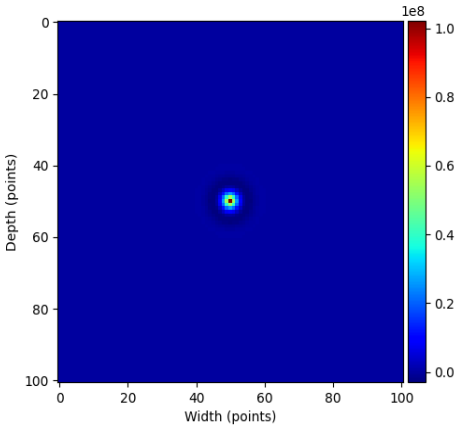
Gráfico 6: Efeito da variação do tamanho do grid



Fonte: Elaborado pelo autor

5.4 DISCUSSÃO DOS RESULTADOS

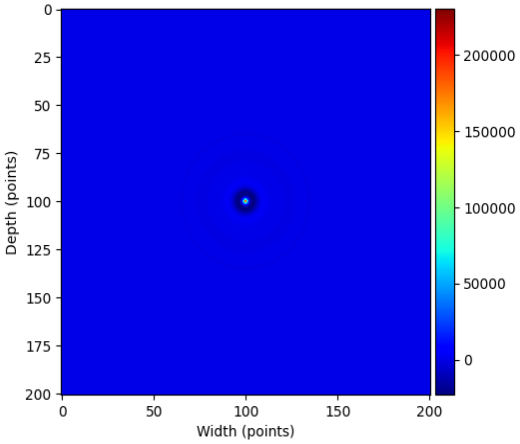
Figura 6: Visualização do gradiente calculado de um grid de 50³ pontos com 500 timesteps



Fonte: Elaborado pelo autor

Como evidenciado anteriormente, o ganho de desempenho no uso de multi-gpus ao se realizar o cálculo do gradiente para um FWI se faz visível somente em problemas com um tamanho considerável de domínio. Essa característica é atribuída ao *overhead* presente na alocação e preenchimento dos vetores de dados nas GPUs, cujo custo computacional cresce linearmente com o tamanho e é constante em relação ao número de iterações; enquanto que o custo computacional do cálculo do gradiente, e consequentemente, os possíveis ganhos de desempenho,

Figura 7: Visualização do gradiente calculado de um grid 1000³ pontos com 500 timesteps



Fonte: Elaborado pelo autor

crescem exponencialmente em relação ao tamanho do grid e linearmente em relação ao número de iterações.

Os resultados obtidos também indicam uma necessidade de aprimoramento na técnica de compressão de dados utilizada pelo projeto, do ponto de vista de desempenho. Vale observar que, apesar de não haverem ganhos de desempenho ao utilizarmos a compressão de dados, a mesma não foi afetada e as possíveis melhorias no uso de dados podem ser melhor estudadas em momento futuro.

Ao compararmos a Tabela 3, que demonstra os resultados obtidos, com a Tabela 2, obtida através de trabalhos anteriores, podemos ver que a estratégia de implementação *multi-gpu* adotada pode obter o resultado esperado, com ganhos de desempenho obtidos em condições similares.

6 CONCLUSÃO

Neste trabalho, foi proposto, implementado e validado um método eficiente para a utilização de técnicas de checkpointing e compressão de dados em um ambiente *multi-GPU* para a resolução do problema de inversão de forma de onda completa. A estratégia adotada visou otimizar a eficiência computacional por meio da decomposição de domínio e da gestão explícita de memória e transferência de dados entre GPUs, reduzindo o tempo de execução e a demanda por memória.

Os resultados experimentais demonstraram que a implementação *multi-GPU* oferece ganhos de desempenho significativos apenas para problemas de grande escala, com um alto número de iterações e domínios de grande tamanho. Para problemas de menor escala, o overhead associado à alocação e transferência de dados entre os dispositivos acaba por mitigar os benefícios do paralelismo distribuído. Ademais, verificou-se que o uso da tecnologia NVLink foi essencial para minimizar os custos de comunicação entre GPUs e que a compressão de dados mostrou-se eficaz para reduzir a demanda de memória sem impactar significativamente o tempo de execução.

A investigação também revelou que a variação do *Space Order* não influenciou consideravelmente o desempenho das implementações, o que pode ser atribuído ao baixo volume de dados trocados entre GPUs durante a execução do método. Em contrapartida, o número de *timesteps* mostrou-se um fator crítico na determinação da eficiência do paralelismo, evidenciando que a abordagem *multi-GPU* é vantajosa apenas para execuções prolongadas.

Dessa forma, este trabalho contribui para o avanço de técnicas de computação de alto desempenho aplicadas a problemas sísmicos, fornecendo diretrizes para a escolha da melhor configuração computacional de acordo com a escala do problema. Como trabalhos futuros, sugere-se a investigação de novas abordagens para reduzir o *overhead* de transferência de dados entre GPUs, bem como a implementação de estratégias adaptativas que otimizem dinamicamente a execução com base nas características do problema em análise.

6.1 TRABALHOS FUTUROS

Embora os resultados aqui obtidos tenham se mostrado significativos, o ganho de desempenho ainda pode ser melhorado. A integração de *multi-gpus* e o cálculo do gradiente do FWI para grids pequenos ou de baixa resolução temporal não ofereceu grandes ganhos de desempenho, justificando que se sejam exploradas em momentos futuros outras abordagens que melhor aproveitem os recursos computacionais disponíveis. Ficam como sugestões diretas de trabalho futuro:

1. **Otimização da transferência de dados e melhor decomposição dos vetores utilizados em cada GPU**: Apesar dos esforços feitos, a organização dos dados nos vetores no código original impediu os dispositivos de GPU carreguem somente os dados necessários em memória para todos os vetores utilizados. Para otimizar o uso de memória e diminuir o *overhead* da cópia inicial, é preciso repensar a estrutura dos vetores de velocidade e *source points*.
2. **Melhora do uso da compressão de dados:** A abordagem desenvolvida anteriormente em uso no FWI em questão fica aquém das expectativas. Sugerimos que diferentes ferramentas e abordagens sejam estudadas em momento futuro para melhorar tanto o ganho de desempenho quanto a interação entre compressão de dados e a decomposição de domínio em *multi-gpus*.
3. **Aplicação em casos reais:** O cálculo do gradiente, cujas etapas foram otimizadas durante o desenvolvimento do presente trabalho, foi até o momento validado em isolamento contra as implementações de *baseline*, deixando em falta a validação com a execução de um FWI completo com todas as variações de parâmetros aqui realizadas. Sugere-se que resultados da execução completa sejam estudados para melhor compreender os ganhos de desempenho e as falhas da abordagem desenvolvida.

REFERÊNCIAS

EMEL'YANOV, V. N.; KARPENKO, A. G.; KOZELKOV, A. C.; TETERINA, I. V.; VOLKOV, K. N.; YALOZO, A. V. Analysis of impact of general-purpose graphics processor units in supersonic flow modeling. **Acta Astronautica**, v. 135, p. 198-207, 2017. DOI: http://dx.doi.org/10.1016/j.actaastro.2016.10.039. Acesso em: 02/12/2024.

FABIEN-OUELLET, G.; GLOAGUEN, E.; GIROUX, B. Time-domain seismic modeling in viscoelastic media for full waveform inversion on heterogeneous computing platforms with OpenCL. **Computers & Geosciences**, v. 100, p. 142-155, Mar., 2017. DOI: 10.1016/j.cageo.2016.12.004. Acesso em: 01/01/2025.

FREIRE, Y. N.; GOMI, E.; SENGER, H. Data mapping strategies for multi-GPU implementation of a seismic application. In: **SIMPÓSIO SOBRE SISTEMAS COMPUTACIONAIS DE ALTA DESEMPENHO (SSCAD)**, 2023, Brasil. Anais [...]. Sociedade Brasileira de Computação (SBC), 2023. Disponível em: https://sol.sbc.org.br/index.php/sscad/issue/view/1189. Acesso em: 02/02/2025.

HENNESSY, J. L.; PATTERSON, D. A. **Computer architecture: a quantitative approach.** San Francisco, CA: Morgan Kaufmann Publishers Inc., 2011.

KUKREJA, N.; HÜCKELHEIM, J.; LOUBOUTIN, M.; WASHBOURNE, J.; KELLY, P.; GORMAN, G. Lossy checkpoint compression in full waveform inversion. **Physics.comp.** v.1, p.1-30. Set., 2020. DOI: 10.5194/gmd-2020-325. Acesso em: 14/08/2024.

MOORE, G. E. Cramming more components onto integrated circuits. Electronics, abr. 1965. **Proceedings of the ieee**, v. 86, n. 1, P. 82-85, Jan. 1998. Disponível em: https://www.cs.utexas.edu/~fussell/courses/cs352h/papers/moore.pdf. Acesso em: 14/05/2024.

PANDEY, M.; FERNÁNDEZ, M.; GENTILE, F.; ISAYEV, O.; TROPSHA, A.; STERN, A. C.; CHERKASOV, A.. The transformational role of GPU computing and deep learning in drug discovery. **Nature Machine Intelligence**, v. 4, n. 3, p. 211-221, Mar. 2022. DOI: 10.1038/s42256-022-00463-x. Acesso em: 29/02/2024.

PONCE, M.; SPENCE, E.; VAN ZON, R.; GRUNER, D. Scientific computing, high-performance computing and data science in higher education. **The Journal of Computational Science Education**, v. 10, n. 1, p. 24–31, 2019. DOI: 10.22369/issn.2153-4136/10/1/5.

SERPA, M. S.; PAVAN, P. J.; CRUZ, E. P.; MACHADO, R. L. M.; PANETTA, J.; AZAMBUJA, A.; CARISSIMI, A.; NAVAUX, P. O. A. Energy efficiency and portability of oil and gas simulations on multicore and graphics processing unit architectures. **Concurrency and Computation: Practice and Experience**, v. 33, n. 18, p. 169-180, Set., 2021. DOI: https://doi.org/10.1002/cpe.6212. Acesso em: 07/08/2024.

SOUZA, J. F. de; MOREIRA, J. B. D.; ROBERTS, K. J.; GAIOSO, R. di R. A.; GOMI, E. S.; SILVA, E. C. N.; SENGER, H. simwave – A Finite Difference Simulator for Acoustic Waves Propagation. **Cornell Univsity** [online], p. 3-32. Jan, 2022. DOI: Disponível em: http://arxiv.org/abs/2201.05278. Acesso em: 08/11/2024.

SYMES, W.. Reverse time migration with optimal checkpointing. **Geophysics**, v. 72, n.5, p. 213-221, Ago., 2007. DOI: https://doi.org/10.1190/1.2742686. Acesso em: 12/03/2024.

KIRK, D. B.; HWU, W. **Programming Massively Parallel Processors**: A Hands-on Approach. 3. ed. Burlington: Morgan Kaufmann, 2016.

NVIDIA. **CUDA C Programming Guide**. NVIDIA Corporation, 2020. Disponível em: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Acesso em: 12/11/2024.

SANDERS, J.; KANDROT, E. **CUDA by Example**: An Introduction to General-Purpose GPU Programming. Waltham: Addison-Wesley, 2010.

LAKATOS, E. M.; MARCONI, M. de A. **Fundamentos de metodologia científica**. 5. ed. São Paulo: Atlas, 2003

SOUZA, Jaime F. de; SENGER, H.; ROBERTS, K.; JOAO-BAPDM; ROUSSIAN. simwave. HPCSys-Lab/simwave: v1.0 (v1.0). **Zenodo**, 2022. DOI: https://doi.org/10.5281/zenodo.5847018. Disponível em: https://github.com/HPCSys-Lab/simwave Acesso em: 15/11/2024.

FICHTNER, A**.** Full Seismic Waveform Modelling and Inversion. In: Advances in Geophysical and Environmental Mechanics and Mathematics. **Springer Berlin Heidelberg**, 2011. Disponível em: https://link.springer.com/10.1007/978-3-642-15807-0. Acesso em: 15/11/2024.

VIRIEUX, J.; OPERTO, S. An overview of full-waveform inversion in exploration geophysics. **Geophysics**, v. 74, n. 6, p. WCC1–WCC26, 2009. DOI: https://doi.org/10.1190/1.3238367.

NVIDIA. *NVLink Bridges*. Disponível em: https://www.nvidia.com/pt-br/design-visualization/nvlink-bridges/. Acesso em: 9 jan. 2025.

NICOLAU, Yuri**.** YuriNicolau/TCC: TCC. Zenodo, 2025. Disponível em: https://doi.org/10.5281/zenodo.14976513. Acesso em: 5 mar. 2025.