Data mapping strategies for multi-GPU implementation of a seismic application

Yuri Nicolau Freire[1], **Edson Gomi** [2], **Hermes Senger** [1]

[1]Departamento de Computação
Universidade Federal de São Carlos (UFSCar) – São Carlos, SP – Brazil
yurinicolau@estudante.ufscar.br, hermes@ufscar.br

[2]Escola Politécnica
Universidade de São Paulo (USP) – São Paulo, SP – Brazil
gomi@usp.br

Abstract. *Seismic imaging applications are computationally costly, and the industry's demand is continuously increasing due to the availability of better data, larger data, and the need for better resolution images. It means that the computational capacity needed tends to increase both in terms of FLOPS calculation and memory. Nowadays, many HPC clusters have nodes with multiple GPUs (e.g., 2, 4, and 8). In this paper, we investigate mechanisms and strategies for the data exchange (of the halo zones) of a finite differences grid of a wave simulator implemented in OpenMP. We compare the performance and programming effort of four data mapping mechanisms supported by OpenMP and CUDA. Our best strategy has achieved speedups of 3.87 on four V100 GPUs with NVLink.*

1. Introduction

Seismic imaging applications like the *full-waveform inversion* (FWI) and the *reverse time migration* (RTM) are extensively used in geophysical exploration for the identification and characterization of hydrocarbon reservoirs. Such applications are extremely important for the efficiency of oil and gas exploration, such as in the Brazilian coastal region where hydrocarbon reservoirs are found a few kilometers deep under salt bodies. One single well drilled in the wrong location can waste millions of US dollars and delay production for weeks or months. Even after the production starts, it is important to track how reservoirs evolve (i.e., the quantities of remaining hydrocarbon, the flow, and pressure of fluids in porous rocks, where to inject fluids to increase pressure, etc) to maximize production.

High-performance computing (HPC) is widely recognized as a major enabling technology of large industrial FWI workflows for oil and gas exploration [Virieux and Operto 2009]. Around the late 1990s and early 2000s, large 3D acoustic FWI to explore oil and gas became feasible due to large HPC clusters [Bohlen 2002]. Over the last decade, clusters powered with GPU accelerated FWI in 3.5-4 times over homogeneous implementations [Gokhberg and Fichtner 2016, Kim et al. 2012].

Despite noticeable advances, FWI and RTM workflows still remain computationally expensive, and the challenges tend to increase in the coming years. Typically, the execution of an *acoustic* FWI takes weeks to months on a multi-Petaflop/s cluster with data that are collected with frequencies between 2 and 10 Hz [Hölzle 2017]. As the acquisition technologies improve, better data with additional information becomes available, increasing computational costs. The industry pushes the demand for higher resolution images

(i.e., processing higher frequency data) to shorten the "time to first oil" and improve exploration efficiency. Techniques for improving the quality of imaging include using better (i.e., higher frequency) data, longer offsets (i.e., longer recordings), including anisotropy, or using better physics (like elastic, visco-elastic or TTI modeling) with more unknowns, among others. All of these improvements imply either more floating-point operations or memory footprint [Virieux and Operto 2009]. As of today, many HPC clusters have heterogeneous nodes with one or more CPUs, and multiple (typically 2, 4, 6 or 8) GPUs. In this scenario, it is mandatory to optimize the applications to benefit from this architecture.

The computational cost of FWI is dominated by the numerical solution of the *forward* and *adjoint wave propagation*, which involve the numerical solution of partial differential equations (PDEs) that model the propagation of acoustic waves in multi-layer materials. In this paper we focus on the parallel implementation of a finite differences acoustic wave solver on multi-GPU heterogeneous nodes. In multi-GPU scenarios, the exchange of halo zones between GPUs can raise the communication overhead and hinder performance [Micikevicius 2009, Gokhberg and Fichtner 2016, Meng and Skadron 2009].

OpenMP has been a staple in parallel programming for years, thanks to its ability to support high performance and productivity for accelerating applications employing a directive-based programming model. Code offloading for accelerators was introduced in its version 4.0, and compiler technology is mature for GPU code offloading. In this paper, we elaborate on previous work which optimizes the application code for single-GPU systems [de Souza et al. 2022c, de Souza et al. 2022b]. The main objective of the present work is to investigate mechanisms for efficient data mapping and exchanging across the memories of an HPC node with multiple GPUs, as this is the first step when moving from single- to multi-GPUs optimization. Device code offloading was introduced in OpenMP 4.0 with the inception of the *target* directive. The native OpenMP data mapping model aims to deliver seamless performance and simplicity as seen in CPU-based parallelization for manycore devices (such as GPUs) by supporting the *target data map* pragma for easy data migration between host and Device. Although these directives allow straightforward programming of single-GPU implementations, its use in multi-GPU systems remains underpowered for applications that require sections of an array to be accessed by different devices. For this reason, synchronization and data consistency of halo regions must be treated, which can be implicitly handled by technologies such as NVidia's Unified Virtual Addressing, or explicitly by using CUDA's or OpenMP's device data functions.

As the main contribution, we compare the performance and programming effort of four strategies that use four data mapping mechanisms, which include (i) the OpenMP's `target` construct data mapping, (ii) the CUDA's Unified Virtual Addressing (UVA), (iii) the CUDA explicit memory copy, and (iv) the OpenMP explicit memory copy. As a general remark, we found that, although implicit data mapping methods can facilitate the programming task on multi-GPUs, explicit memory copy methods can support better scalability for our seismic application running on multi-GPUs. With the aid of explicit methods, we could obtain speedups of up to 3.87 on four GPUs.

The rest of this paper is organized as follows: related works are discussed in Section 2. Section 3 briefly presents our seismic application; Section 4 presents our data mapping strategies for multi-GPU nodes. Section 5 presents experimental evaluation of our strategies. Finally, Section 6 present our conclusion and directions for future work.

2. Related Works

Around the late 1990s and early 2000s, the execution of large 3D acoustic FWI for oil and gas exploration became feasible on large HPC clusters [Bohlen 2002]. Over the last decade, heterogeneous clusters have proven to accelerate FWI in 3.5-4 times over homogeneous implementations [Gokhberg and Fichtner 2016, Kim et al. 2012].

A previous work investigated the performance of an acoustic wave solver on modern GPUs using OpenMP [de Souza et al. 2022b]. This work aims to investigate several loop transformations introduced in OpenMP for GPUs and implemented in recent compilers to reach maximum performance on single GPUs. The parallelization of the application using the OCCA library has been addressed in [de Souza et al. 2022a]. OCCA is a library that can produce optimized code for different architectures such as CPUs and GPUs. The objective of this work is not limited to evaluating the maximum performance that could be obtained but to assess the ability of OCCA to support performance portability for high-order stencil codes like those implemented in Simwave on different GPU architectures.

[Gokhberg and Fichtner 2016] implemented a spectral-element-based FWI for large heterogeneous HPC systems, applied for the 3D imaging of large regions. The work focused on finding the optimal parallelization configurations of individual simulations, the large I/O requirements of adjoint simulations, and the scheduling of large numbers of forward and adjoint solves. In another work, [Cai et al. 2018] proposed strategies to implement a finite differences-based RTM on GPUs. They also optimize the checkpointing to balance the trade-off between memory requirements and recomputations.

Seismic applications are compute-bound and manage vast amounts of data [Datta et al. 2018]. This work proposed an RTM algorithm that takes advantage of the fast NVLink interconnect and Non-Volatile Memory Express (NVMe) memory in a cluster with four GPUs per node. They proposed a two level checkpointing for optimizing memory usage, computations and data movement.

The works discussed so far have a standard limitation. They provide implementations of seismic applications on single-GPU systems. We believe efficient implementations on multi-GPUs are considerably more challenging, but this is a necessary step ahead for many reasons. First, as mentioned the industry continuously demands better-resolution images, which increases the amount of floating-point operations and memory usage. Nowadays, many HPC clusters have multi-GPU nodes, with 2, 4, 6 or 8 GPUs. In this scenario, it is mandatory to optimize the applications to benefit from this architecture, thus reducing the computation times of seismic applications and allowing the processing of higher-resolution grids, potentially larger than the memory of a single GPU.

[Deng et al. 2021] proposed a 3D electromagnetic full waveform inversion workflow in multi-GPU systems. The domain is divided into 2, 4 or 8 subdomains processed by the GPUs. GPUs in the same node exchange halo data through P2P mechanisms and MPI for GPUs in different nodes. Although this work explores multi-GPU parallelism, we have different approaches. First, we want to avoid halo data exchanges between GPUs in different nodes, which raises inter-node communication. Second, by keeping all the subdomains on GPUs in the same node we avoid inter-node communication due to the saving and retrieval (in reverse order) of snapshots during the adjoint phase to calculate the gradients. Therefore, our paper focuses on studying mechanisms and strategies to

exchange halo data among GPUs within the same node.

3. The seismic imaging application

Our seismic application kernel is part of the *full-waveform inversion* (FWI). FWI is a data-fitting procedure based on full-wavefield modeling to extract quantitative information from collected data. FWI is widely used in seismology and geophysical exploration to build high-resolution images from subsurface materials based on their physical properties. In conventional FWI workflows, most of the time is spent in the computation of the *forward* and *adjoint wave propagation*. The kernel of this process involves the numerical solution of partial differential equations (PDEs) that model the propagation of acoustic waves in multi-layer subsurface materials. The FWI workflow minimizes both amplitude and phase differences between the signals that are recorded with a set of microphones (or hydrophones) located near the surface. The model is incrementally modified so that the functional that represents the error is sufficiently reduced [Virieux and Operto 2009]. The final objective is to reconstruct various parameters of the materials, such as *the velocities of P-waves and S-waves, density, anisotropy, and attenuation*. The wave propagation kernel is also part of the *reverse-time migration* (RTM) application, which is a quite similar process that uses least-squares minimization of the misfit between recorded and modeled data. One difference between RTM and FWI is that the seismic wavefield recorded at the receiver is back propagated in reverse time migration, whereas the data misfit is back propagated in the waveform inversion.

3.1. The wave propagation kernel

There are various wave equations, based on different physic models. For this work, we use the simplified acoustic wave equation, assuming an isotropic medium, constant density, and neglecting shear strains as defined in [de Souza et al. 2022c]. This is a second-order differential equation that describes particle displacement, as follows:

$$\frac{\partial^2 p}{\partial t^2}(x,t) - c^2(x)\nabla^2 p(x,t) = -\rho c^2(x)\nabla . b(x,t) \tag{1}$$

where ρ corresponds to the density, b are external body forces, p represents scalar pressure, $c = \sqrt{\frac{k}{\rho}}$ is the wave speed and $\nabla = (\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z})$ in cartesian coordinates. Using the finite differences method, Equation 1 can be discretized using second-order in time and variable spatial order (even, varying from 2 to 20). Finally, the product $\nabla \rho_{i,j,k} \nabla p^n_{i,j,k} = 0$, and the equation reduces to a stencil calculation described as:

$$\begin{aligned} p^{n+1}_{i,j,k} &= 2p^n_{i,j,k} - (1 - \eta\Delta t)p^{n-1}_{i,j,k} \\ &+ \frac{c^2\Delta t^2}{1+\eta\Delta t}\frac{1}{\Delta x^2}(v_0 p^n_{i,j,k} + \sum_{m=1}^{r} v_i(p^n_{i+m,j,k} + p^n_{i-m,j,k})) \\ &+ \frac{c^2\Delta t^2}{1+\eta\Delta t}\frac{1}{\Delta y^2}(v_0 p^n_{i,j,k} + \sum_{m=1}^{r} v_i(p^n_{i,j+m,k} + p^n_{i,j-m,k})) \\ &+ \frac{c^2\Delta t^2}{1+\eta\Delta t}\frac{1}{\Delta z^2}(v_0 p^n_{i,j,k} + \sum_{m=1}^{r} v_i(p^n_{i,j,k+m} + p^n_{i,j,k-m})). \end{aligned} \tag{2}$$

Finally, the discretized form of the wave equation can be computationally solved as in Algorithm 1. The values accessed when computing each cell when using stencil radius of 1 is exemplified in Figure 1.

Algorithm 1 Sequential wave propagation algorithm

```
1: for iteration ∈ [0, n] do
2:     for k ∈ [stencil_radius, size_z − stencil_radius] do
3:         for j ∈ [stencil_radius, size_y − stencil_radius] do
4:             for i ∈ [stencil_radius, size_x − stencil_radius] do
5:                 Compute p^{n+1}_{i,j,k} as described in Eq. 2
6:             end for
7:         end for
8:     end for
9: end for
```

3.2. The Simwave solver

We used the wave equation simulator implemented by Simwave [de Souza et al. 2022c], a Python/C package that enables researchers to model acoustic waves propagation. Simwave is distributed as open software [1]. The package is comprised of modular back-end kernels written in C, which provides both a numerically accurate solver of the acoustic wave equation for Geophysicists and a realistic application kernel for HPC researchers. Simwave has been used in previous research for single GPUs implementations and optimization, as in [de Souza et al. 2022b, de Souza et al. 2022a].

3.3. Moving data between multi-GPUs

In this work, our baseline is an OpenMP implementation of the wave simulation on single GPUs presented in [de Souza et al. 2022b]. Our objective is to study strategies for efficient implementation of the acoustic wave solver for heterogeneous single-node systems composed of multiple GPUs. The potential benefit is twofold: (i) to reduce the execution times of the application on multi-GPU nodes; and (ii) to allow the execution of finite difference grids whose size exceeds the memory of single GPUs. The parallelization for GPUs placed on different nodes, however, is beyond this work's scope.

Our baseline code implements the acoustic wave solver on single GPUs using OpenMP. The data is copied to the device memory at the beginning of the target region. Within the target region, the spatial loops for the stencil calculation are parallelized as described in [de Souza et al. 2022b]. At the end of the target region, results are moved to the host memory. The process is straightforward and there is no need for explicit data allocation and manipulation [Mattson et al. 2019].

Algorithm 2 provides an abstract view of our multi-GPU implementation. Line 1 computes the subdomain bounds by partitioning the matrix in the z axis (because y and x axis are closer in memory than z). Line 3 distributes the subdomains for parallel execution on multiple devices. Line 3 assigns one subdomain for each device to process, and Lines 4 to 10 compute the stencil inside each subdomain. A point of attention is the exchange

[1] `https://github.com/HPCSys-Lab/simwave.git`

of the halo regions (i.e., data elements shared between neighbor GPUs) at the end of each time step, as illustrated in line 11 of Algorithm 2.

One drawback of OpenMP target data map is the lack of support for overlapping array sections (i.e., the halo zones) on multiple devices. One possible solution is mapping the entire arrays onto all devices, with the halo regions explicitly updated. However, this strategy prevents us from taking advantage of the extended memory pool and from using peer-to-peer communication mechanisms. Another solution involves explicitly allocating and copying the arrays, which may increase complexity. A second approach consists in using the CUDA runtime library to explicitly handle data movement between the devices. These approaches will be investigated in the coming sections.

Algorithm 2 Multi-GPU wave propagation algorithm

```
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
```

4. Data mapping strategies

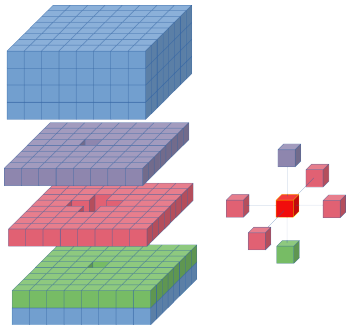

Figure 1. Data required for processing a cell using a radius 1 stencil

This section proposes four strategies for implementing the wave propagation solver on multi-GPU systems. Our implementations aim to compare the complexity and performance of different domain decomposition strategies using either CUDA's runtime library or OpenMP's native device data transfer functions.

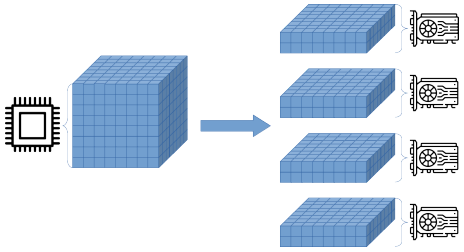

Figure 2. Visual representation of domain decomposition for four gpus

4.1. OpenMP native data mapping

This strategy uses the OpenMP target data map constructs [Xu et al. 2015] for domain decomposition. Although it features implicit data transfers into the devices before execution, updating the halos between iterations must be done explicitly. It is built directly upon the baseline implementation by creating one host thread for each device, which is responsible for defining its starting and ending indexes and initiating the GPU kernel execution.

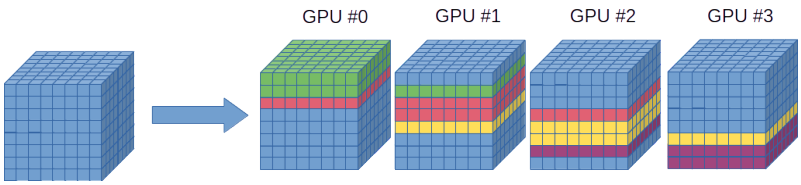

Figure 3. OpenMP domain decomposition

Using `omp target data map` to map data and `omp target update` to exchange halo data between devices presents some drawbacks. First, it does not support mapping overlapping array sections (e.g., halo data). To allow multiple devices to access shared data sections, the entire array must be mapped onto every GPUs in the node. This requires every device to have enough memory to house the entire array (as depicted in Fig. 3), thus wasting memory space. Second, the halo data exchanges must be managed explicitly, which makes the process more error-prone. Finally, current implementations of these mechanisms (by the compilers) does not take advantage of direct communication between devices. Therefore, OpenMP native data mapping is unsuited for this application, which exchanges halo data in each iteration.

4.2. Unified Virtual Addressing

Our second implementation uses the CUDA's Unified Virtual Addressing (UVA) for manipulating data across multiple GPUs . This strategy aims to improve performance while maintaining low complexity. We use the `cudaMallocManaged` function (implemented by the CUDA runtime library) for allocating data in a unified space between the host and the devices. As memory allocation and copy are handled by the CUDA runtime driver, all devices can access the same pointers, and no explicit updates are necessary between iterations. We implemented data prefetching at the end of each iteration to avoid page faults and on-demand data copy, thus improving performance.

Besides data prefetching, preferred location functions were added wherever necessary, and copies of read-only arrays used in the computation were explicitly allocated

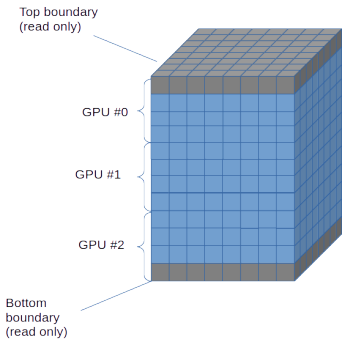

Figure 4. UVA domain decomposition

in each device to ensure local access. As in the previous implementation, one host thread was created for each device, to calculate the starting and ending points for each subdomain, and launching the GPU kernel execution. To allow UVA pointers to be interpreted by the OpenMP kernels using the *target* directive, the `is_device_ptr` clause was used.

4.3. Explicit copy using Auxiliary Halos

Our third implementation introduces explicit memory allocation and transfer. This method involves the allocation and initialization of grid slices on each device, as well as separate arrays for top and bottom halo data as illustrated in Fig. 5. This strategy allows overlapping the processing of inner grid cells and halo updates. For this implementation, we also compare the performance between CUDA's native `cudaMalloc()` and `cudaMemcpy()`, and OpenMP's `omp_target_alloc()` and `omp_target_memcpy()` explicit device data management functions.

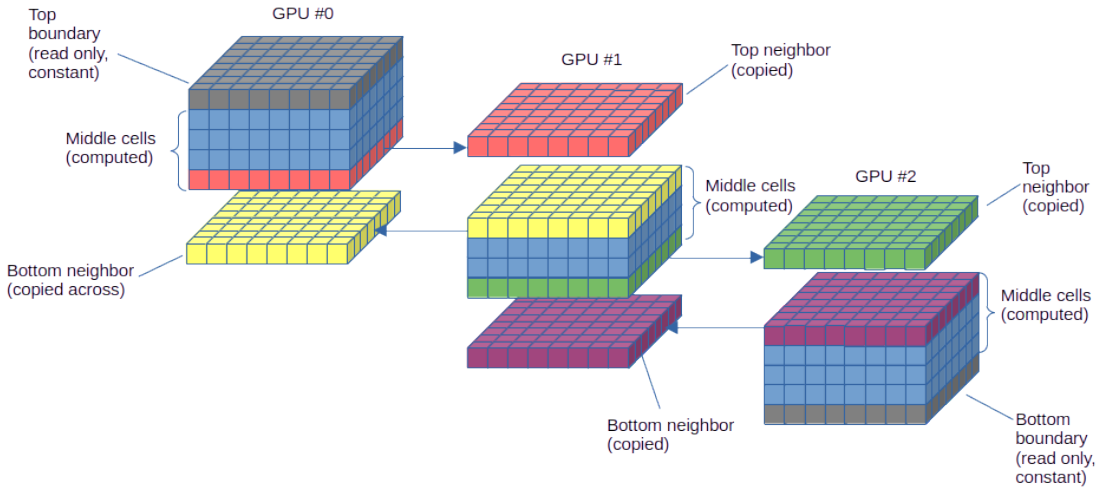

Figure 5. Visual representation of the explicit data copy with auxiliary halos.

Similar to in the previous strategy, we created one host thread per device, to allocate and initialize the data in the device. Since the neighbor data is copied into auxiliary arrays, the computation is split into three blocks: (i) the inner cells processing (overlapped with the halo data prefetching); (ii) computation of the top border; and (iii) computation of the bottom border. After the processing, the update of neighboring copies is initiated

and a new iteration begins. This method requires the use of the `is_device_ptr` clause to allow the OpenMP kernel to use the device memory pointers.

Splitting data into three arrays, however, proved much too complex for deployment in the full version of Simwave, especially for supporting stencil radii bigger than 1. Since the neighboring cells may be located on either the main or auxiliary arrays, as exemplified in figure 6, a condition must be implemented to specify which memory address should be utilized, for both top and bottom loops. Translating the array position between local and auxiliary arrays also demands a lot of attention, introducing a lot of complexity and decreasing code readability.

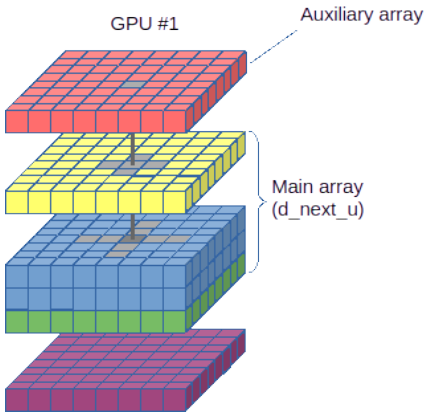

Figure 6. Auxiliary array access for a stencil radius of 2

4.4. Explicit copy using Coupled Halos

This implementation builds directly upon the previous, with explicit memory allocation and data transfers. However, each device's array was increased as to accommodate their own copy of the halo, and a single loop was implemented instead of three distinct ones. Just like in the previous one, this strategy was implemented in two ways - using CUDA's `cudaMalloc()` and `cudaMemcpy()`, and OpenMP's `omp_target_alloc()` and `omp_target_memcpy()` functions.

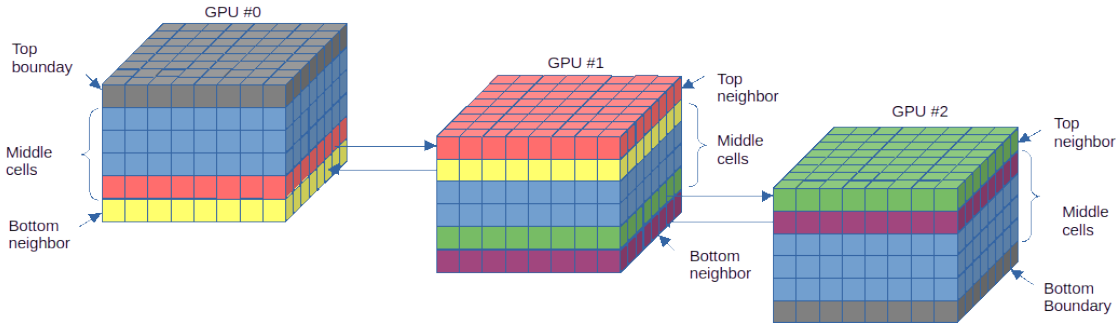

Figure 7. Domain decomposition for the Coupled Halos implementation

For data initialization, the height of data blocks are increased by $2 \times n$ (where n is the stencil radius). The starting and ending positions are calculated by the host threads and vary depending on which section of the array is being processed since the top and bottom

SO	Grid size	# of iters.	Baseline (1 GPU) Time	OMP Data map Time	OMP Data map S	UVA Time	UVA S	Auxiliary OMP Time	Auxiliary OMP S	Coupled OMP Time	Coupled OMP S	Auxiliary CUDA Time	Auxiliary CUDA S	Coupled CUDA Time	Coupled CUDA 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

Table 1. Execution times (seconds) and speedup (S) for all strategies processing 3D grids for 4 GPUs. Speedups above 3.5 are highlighted in dark green, light green ($3 < S \leq 3.5$), lime ($2.5 < S \leq 3$), yellow ($2 < S \leq 2.5$), and red ($S < 2$).

slices need not compute their own top and bottom borders. The kernel is similar to the baseline code for single-GPU, except for the calculation of starting and ending positions, the halo updates between iterations, and the `is_device_ptr()` clauses.

5. Experiments

We tested our implementations on a server with 2 Intel Xeon Gold 6130, 192 GB DDR4, 4 NVIDIA Tesla V100 SXM NVLink with 5,120 CUDA cores and 32GB of HBM2 memory. The code was compiled using LLVM's Clang compiler, with OpenMP version 4.5 and CUDA version 11. The cluster uses SLURM for job management and Singularity containers, with kernel version 5.12 and hosted on a CentOS linux operating system.

Table 1 displays the execution times taken as an average of 3 runs, for all multi-GPU implementations of the acoustic wave solver. The first column presents the execution times for the baseline (i.e., single GPU implementation). We used 3D grids of 256^3,512^3, and 1024^3 FP32 cells. The solver used spatial discretizations with 2nd, 4th, and 8th spatial orders. The simulations tested have 400 and 4000 iterations. Each cell's color reflects that implementation's performance, with green representing high speedup (> 3) and red, low speedup (< 2). Our best strategies achieved speedups of 3.7 (Coupled OMP), 3.85 (Auxiliary CUDA), and 3.8 (Coupled CUDA) on a four-GPUs system.

As displayed in table 1, larger data sets allow for higher speedups, owing to the lower impact of communication overheads. Likewise, the impact of overlapping processing and data movement as enabled by the *Auxiliary* methods diminishes as the overall processing time increases.

5.1. Final remarks

In summary, the implicit methods tested (OpenMP data map and CUDA Unified Virtual Addressing) yield unimpressive results from an execution time perspective, barely surpassing 2x speedup for larger number of iterations on larger grid sizes. While Unified

Virtual Addressing might be recommended for increasing memory capacity and small performance gains, the OpenMP `target data map` is underpowered for multi-GPU implementations of stencil-style algorithms, and thus is not recommended.

We tested two strategies for explicitly managing the allocation and data copy: (i) using auxiliary arrays for the halo manipulation, and (ii) increasing the size of data slices on each device to accommodate the halos and reduce complexity. Both strategies lead to lower execution times when compared to implicit methods, while maintaining minimal memory usage. Despite their increased complexity, the performance of explicit allocation and copy functions justify their use over their implicit counterparts. Comparing both explicit strategies, despite the use of auxiliary arrays for the halo enabling better overlapping of data copy and computation, the benefits of such were inconsistent, while demanding more complex and error-prone implementation. On the other hand, using a single array presents better readability with similar performance.

When comparing CUDA's `cudaMalloc` and `cudaMemcpy` functions to OpenMP's `omp_taget_alloc` and `omp_target_memcpy`, the former consistently performs better for all combinations of grid sizes, stencil radii and number of iterations, being a favorite for deployment and further optimization.

6. Conclusion

Seismic applications present huge computational costs and the industry demands increasingly better and faster-resolution applications. GPUs have demonstrated outstanding performance in the execution of seismic applications. Efficient single GPU implementations are currently deployed on state-of-the-art industry seismic workflows. The current need for increasing the performance of seismic applications allied to the availability of multi-GPU architecture HPC clusters raises the urgent need to develop efficient implementations on multi-GPU systems.

In this paper, we studied four data manipulation strategies focusing on the efficiency and implementation complexity of the halo exchanges on multi-GPU systems. Our best strategies sustained approximate speedups of 3.7 (Coupled OMP), 3.85 (Auxiliary CUDA), and 3.8 (Coupled CUDA) on a four-GPUs system. This is only the first step in the development roadmap for industrial seismic applications, including techniques such as checkpointing and data compression that focus on multi-GPU systems.

Acknowledgements

This work is partially supported by Programa Institucional de Bolsas de Iniciação Científica - PIBIC/CNPq/UFSCar. The authors thank FAPESP (Processes 2019/26702-8 and 2023/00566-6) and Financiadora de Estudos e Projetos (FINEP) for their support.

References

Bohlen, T. (2002). Parallel 3-d viscoelastic finite difference seismic modelling. *Computers & Geosciences*, 28(8):887–899.

Cai, X., Liu, Y., and Ren, Z. (2018). Acoustic reverse-time migration using gpu card and posix thread based on the adaptive optimal finite-difference scheme and the hybrid absorbing boundary condition. *Computers & Geosciences*, 115:42–55.

Datta, D., Appelhans, D., Evangelinos, C., and Jordan, K. (2018). An asynchronous two-level checkpointing method to solve adjoint problems on hierarchical memory spaces. *Computing in Science and Engineering*, 20:39–55.

de Souza, J., Moreira, J., Oliveira, A., Roberts, K., Gomi, E., Silva, E., and Senger, H. (2022a). A GPU accelerated modeling and simulation of acoustic waves for seismic applications. In *Third EAGE Workshop on HPC in Americas*, pages 1–5. European Association of Geoscientists & Engineers.

de Souza, J. F., Machado, L. S. F., Gomi, E. S., Tadonki, C., Mcintosh-Smith, S., and Senger, H. (2022b). Performance of openmp loop transformations for the acoustic wave stencil on gpus. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (Supercomputing 22 - Poster track)*.

de Souza, J. F., Moreira, J. B. D., Roberts, K. J., di Ramos Alves Gaioso, R., Gomi, E. S., Silva, E. C. N., and Senger, H. (2022c). simwave - A finite difference simulator for acoustic waves propagation. *CoRR*, abs/2201.05278.

Deng, J., Rogez, Y., Zhu, P., Herique, A., Jiang, J., and Kofman, W. (2021). 3d time-domain electromagnetic full waveform inversion in debye dispersive medium accelerated by multi-gpu paralleling. *Computer Physics Communications*, 265.

Gokhberg, A. and Fichtner, A. (2016). Full-waveform inversion on heterogeneous hpc systems. *Computers & Geosciences*, 89:260–268.

Hölzle, U. (2017). Schlumberger chooses GCP to deliver new oil and gas technology platform. https://cloud.google.com/blog/topics/inside-google-cloud/schlumberger-chooses-gcp-to-deliver-new-oil-and-gas-technology-platform. Last accessed on 2nd May, 2023.

Kim, Y., Shin, C., and Calandra, H. (2012). 3D Hybrid Waveform Inversion With GPU Devices. SEG Int. Exposition and Annual Meeting. SEG-2012.

Mattson, T., He, Y., and Koniges, A. (2019). *The OpenMP Common Core: Making OpenMP Simple Again*. Scientific and Engineering Computation. MIT Press.

Meng, J. and Skadron, K. (2009). Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus. In *The International Conference on Supercomputing*, pages 256–265.

Micikevicius, P. (2009). 3d finite difference computation on gpus using cuda. In *Workshop on General Purpose Processing on Graphics Processing Units*, page 79–84, New York, NY, USA. ACM.

Virieux, J. and Operto, S. (2009). An overview of full-waveform inversion in exploration geophysics. *GEOPHYSICS*, 74(6):WCC1–WCC26.

Xu, R., Tian, X., Chandrasekaran, S., and Chapman, B. (2015). Multi-gpu support on single node using directive-based programming model. *Scientific Programming*, 2015:621730.