GAPS: GPU-Acceleration of PDE Solvers for Wave Simulation

Bagus Hanindhito
Dimitrios Gourounas
hanindhito@bagus.my.id
dimitrisgrn@utexas.edu
The University of Texas at Austin
Austin, TX, USA

Arash Fathi
Dimitar Trenev
arash.fathi@exxonmobil.com
dimitar.trenev@exxonmobil.com
ExxonMobil Technology and Engineering
Annandale, NJ, USA

Andreas Gerstlauer
Lizy K. John
gerstl@ece.utexas.edu
ljohn@ece.utexas.edu
The University of Texas at Austin
Austin, TX, USA

ABSTRACT

Large-scale simulations of wave-type equations have many industrial applications, such as in oil and gas exploration. Realistic simulations, which involve a vast amount of data, are often performed on multiple nodes of an HPC cluster. Using GPUs for these simulations is attractive due to considerable parallelizability of the algorithms. Many industry-relevant simulations have characteristics in their physics or geometry that can be exploited to improve computational efficiency. Furthermore, the choice of simulation algorithm impacts computational efficiency significantly.

In this work, we exploit these features to significantly improve performance for a class of problems. Specifically, we use the discontinuous Galerkin (DG) finite element method, along with the Gauss-Lobatto-Legendre (GLL) integration scheme on hexahedral elements with straight faces, which then greatly reduces the number of BLAS operations, and simplify the computations to Level-1 BLAS operations, reducing the turn around time for wave simulation. However, attaining peak performance of GPUs is often not possible in these codes that exacerbate bottlenecks caused by data movement, even when modern GPUs enjoying the latest high-bandwidth memory are being used.

We have developed GAPS, an efficient and scalable, GPU-accelerated PDE solver for Wave Simulation, by using hardware- and data-movement-aware algorithms. While significant speed-up over CPUs can be achieved, data movement still limits GPU performance. We present several optimization strategies, including kernel fusion, Look-Up-Table-based neighbor search, improved shared memory utilization, and SM-occupancy-aware register allocation. They improve performance up to 84.15x over CPU implementations and 1.84x over base GPU implementations on average. We then extend GAPS to support multi-GPUs on multi-node HPC clusters for large-scale wave simulations, and perform additional optimizations to reduce communication overhead. We also investigate the performance of several MPI libraries in order to fully overlap communication and computation. We are able to reduce the communication overhead by 70%, and achieve weak-scaling over 128 GPUs.

ICS '22, June 28–30, 2022, Virtual Event, USA

ACM ISBN 978-1-4503-9281-5/22/06…$15.00
https://doi.org/10.1145/3524059.3532373

CCS CONCEPTS

• **Computing methodologies → Massively parallel and high-performance simulations**; *Parallel algorithms*; • **Applied computing** → *Earth and atmospheric sciences.*

KEYWORDS

GPU acceleration, HPC, wave simulation, discontinuous Galerkin, parallel algorithms, optimization strategies

ACM Reference Format:
Bagus Hanindhito, Dimitrios Gourounas, Arash Fathi, Dimitar Trenev, Andreas Gerstlauer, and Lizy K. John. 2022. GAPS: GPU-Acceleration of PDE Solvers for Wave Simulation. In *2022 International Conference on Supercomputing (ICS '22), June 28–30, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3524059.3532373

1 INTRODUCTION

Wave simulations have become one of the important topics in science and engineering, with applications in seismic hazard mitigation [34], medical imaging [13, 26], deriving the composition and features of the Earth's subsurface [19, 22, 29], oceanography [8], or military [16]. We investigate the performance of discontinuous Galerkin (DG) discretizations, when used with the Gauss-Lobatto-Legendre (GLL) integration scheme, on straight-faced hexahedral elements. While the computational cost of these types of wave simulations is considerably lower than those on general meshes, they are still prohibitively expensive for many high-fidelity, industry-relevant applications. Furthermore, reducing the computational costs, in turn, proportionally increases the impact of data movement. With its unique challenges, we explore the performance of the wave computations on state-of-the-art GPUs in this work.

Modern GPUs and emerging architectures can potentially make such high-fidelity simulations feasible [1]. A key challenge in designing efficient algorithms for this type of wave solvers in particular is taming the communication cost between multiple kernels and devices: due to keeping the local computations at a bare minimum, the ratio of communication to computation cost increases, which makes it harder to hide the communication overhead by using conventional strategies, such as overlapping communication and computation.

In this work, we explore strategies to efficiently use GPUs for this type of wave computations, and alleviate some of the key data movement and communication challenges for developing GAPS, an efficient and scalable high-performance GPU-accelerated PDE solver for Wave Simulation. After briefly reviewing some key application areas, the equations to be solved, and discretization algorithms that are better suited on modern architectures, we study how the overhead of data movement between off-chip and on-chip memory limits GPU performance, and implement several dataflow optimizations to reduce data movement and improve performance. We also improve multi-GPU performance by using asynchronous, CUDA-aware MPI communication, which has recently become available. These optimizations lead to significant performance improvements, which can be applied to many similar applications, especially HPC applications that are often memory-bounded [7] due to how the large data is processed. Our contributions include:

- We use hardware-informed algorithms and mesh configurations in developing GAPS for wave simulation in GPUs, to exploit the available parallelism and achieve significant speed-up over CPUs.
- We characterize and identify key bottlenecks of each GPU kernel, based on which we develop advanced dataflow-aware optimization strategies, including kernel fusion, LUT-based neighbor search, improved shared memory utilization, and SM-occupancy-aware register allocation. Our base GPU implementation yields 45.67x average speed-up over CPU, and the optimized GPU implementation yields 84.15x speed-up.
- We extend the scalability of GAPS as a multi-GPU, multi-node wave simulator, investigate the performance of several MPI libraries, and reduce the communication overhead. Our implementation achieves perfect weak-scaling.

2 BACKGROUND

2.1 Full-wavefield Inversion

Full-wavefield inversion [11, 22] is the state-of-the-art method to estimate material properties of the earth's subsurface. The estimated material properties can then enable multiple opportunities, such as finding regions that contain hydrocarbons (e.g., in oil and gas exploration), identifying safe zones for carbon sequestration, as well as finding appropriate locations for storing hydrogen, which is a promising source of low-carbon energy.

Seismic waves are used as probing agents of the subsurface [39] in full-wavefield inversion (Fig. 1-ⓐ). Once waves are artificially generated, typically on the surface of the unknown medium, they propagate through the medium. Due to the heterogeneity of the subsurface medium, these waves change their amplitude and direction of propagation. Some of the reflected waves travel back to the surface (Fig. 1-ⓑ), and are recorded by receivers (Fig. 1-ⓒ). One then seeks to estimate subsurface material properties based on known input waves, known output waves at receiver locations, and a model that describes physics of the wave-motion in the subsurface; this is an iterative process, which requires repeated simulations of the wave-motion [9]. Therefore, fast and efficient wave-solvers are among the crucial elements of this framework.

Due to its high computational cost, full-waveform inversion was not used in realistic settings before the advent of petascale computers. The increase in computational power due to modern and emerging architectures can substantially improve the resolution and fidelity of subsurface images.

2.2 Acoustic and Elastic Wave Equation

The acoustic wave equation is used to model the propagation of compressional waves in fluids, air, earth, body tissues, and many other domains. It is described by partial differential equations:

$$\frac{\partial p}{\partial t} + \kappa \nabla \cdot \mathbf{v} = 0, \tag{1a}$$

$$\frac{\partial \mathbf{v}}{\partial t} + \frac{1}{\rho} \nabla p = 0, \tag{1b}$$

where $p = p(x, y, z, t)$, and $\mathbf{v} = \mathbf{v}(x, y, z, t)$, denote the pressure and velocity fields, respectively; κ and ρ are material properties; $\nabla \cdot \mathbf{v} = \frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z}$ is the divergence of the velocity field, and $\nabla p = \frac{\partial p}{\partial x}\mathbf{i} + \frac{\partial p}{\partial y}\mathbf{j} + \frac{\partial p}{\partial z}\mathbf{k}$ is the gradient of the pressure field.

The elastic wave equation models the propagation of compressional and shear waves within the earth and elastic solids, and is described by:

$$\frac{\partial \mathcal{S}}{\partial t} = \mu(\nabla \mathbf{v} + \nabla \mathbf{v}^T) + \lambda \nabla \cdot \mathbf{v}\, \mathcal{I}, \tag{2a}$$

$$\frac{\partial \mathbf{v}}{\partial t} = \frac{1}{\rho} \nabla \cdot \mathcal{S}, \tag{2b}$$

where $\mathcal{S}$ and $\mathcal{I}$ represent the second-order stress and identity tensors, and λ, μ, and ρ are material properties.

Upon spatial (Fig. 1-ⓓ, ⓔ, ⓕ) and temporal discretization of Equation (1) or Equation (2), a time-marching scheme can be constructed, where solution values at a future time step $(n + 1)$ can be computed based on solution values at a current time step (n):

$$u^{n+1} = Au^n, \tag{3}$$

where u represents the unknown fields that we wish to compute (e.g., $u = \{\mathcal{S}, \mathbf{v}\}$ in the elastic case) [10]. The action of A on u primarily consists of Level-1 BLAS operations when tensor-product hexahedral elements are used.

We remark the electromagnetic wave equations [2], and the Euler equations (not shown) are structurally similar to Equations (1-2), and have similarity in computational schemes [36].

2.3 Discontinuous Galerkin Discretization

There are several methods that are commonly-used for the numerical simulation of the seismic (i.e., acoustic, or elastic) wave equation[1]. These are: the finite difference method (FDM), spectral-element finite element method (SEM), and spectral-element discontinuous Galerkin method (DG). All three methods enable efficient, explicit time-stepping, which helps parallel scalability.

The FDM is the most commonly-used method in practice. It is typically used on a uniform grid, which leads to fast Level-1 BLAS computations. On the downside, it entails higher communication cost than SEM and DG[2]; moreover, it is difficult to develop stable high-order schemes for non-conforming finite-difference grids.

The SEM is typically used in regional seismology, where, due to the large size of the studied region, topography needs to be represented accurately [34]. Unstructured mesh generation could be very challenging and labor-intensive. Furthermore, efficient and accurate fluid-solid coupling is challenging for SEM.

[1] See [35] for a comprehensive review.

[2] Here, we have cross-node HPC communication in mind. While only information at the surface of an element in DG is needed for ghost exchange, deep ghost exchange is needed in FDM when high-order methods are used. High-order methods are necessary to limit the dispersion error [18].

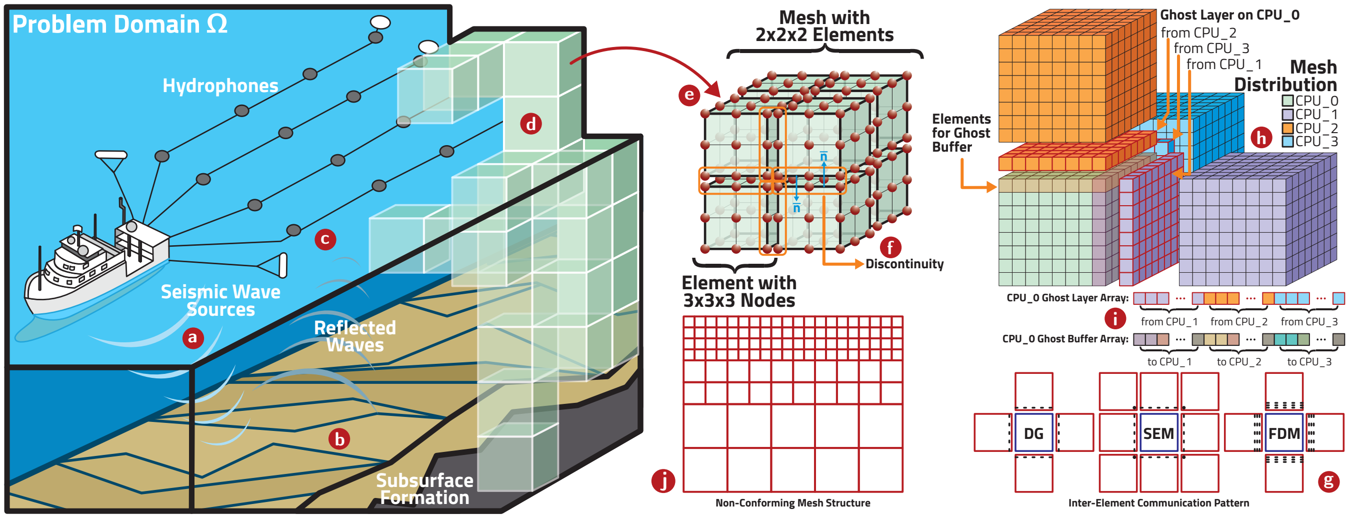

Figure 1: Marine seismic survey is used to collect data for full-waveform inversion by generating seismic waves ⓐ. Reflected waves ⓑ are recorded by hydrophones ⓒ. For computer simulations, the problem domain needs to be discretized into smaller elements ⓓ, each of which has a number of nodes ⓔ. Due to DG discretization, there are discontinuities at the boundaries of the elements ⓕ, and the elements need to communicate with each other to solve them. The inter-element communication pattern comparison between DG, SEM, and FDM is shown in ⓖ. DG only needs information on the face of neighboring elements, SEM needs face and corner information of neighboring elements, and FDM needs face and behind the face information of neighboring elements. To support larger problem sizes, the mesh is partitioned and distributed into multiple CPUs or GPUs ⓗ, where Ghost Exchange is needed ⓘ. A non-conforming mesh can add resolution where it is needed, resulting in computational efficiency ⓙ.

We believe DG [15, 44] has a better chance of harnessing the computing power of modern and emerging architectures: its communication cost is lower than FDM and SEM[3] (Fig. 1-ⓖ) with potential for further reductions; recently, robust and stable algorithms for non-conforming DG discretizations (Fig. 1-ⓙ) have been developed [21]. For many industry-relevant DG wave simulations, Level-1 BLAS operations constitute the bulk of the computations. Therefore, we use DG in this work.

2.4 Gauss-Lobatto-Legendre Integration Scheme

The use of Gauss-Lobatto-Legendre (GLL) [33] integration on straight-faced hexahedral elements reduces the total required BLAS operations, and simplifies those operations into Level-1 BLAS. This way, computing derivatives and interpolations at each spatial direction decouples from other directions, and will rely on a small subset of nodes. This is different than cases that rely on curvilinear, unstructured, or octree meshes that use other integration schemes.

2.5 Data Movement

Although our DG-based discretization yields algorithms that have higher data locality, data movement is still needed. We outline the main forms of data movement in our DG algorithm:

2.5.1 Intra-Device Data Movement. Due to the limited capacity of the on-chip memory [48], the data must be fetched from off-chip memory to perform computations. If not orchestrated carefully, excessive data movement between on- and off-chip memory limits performance and increases energy consumption [41].

2.5.2 Inter-Device, Intra-Node Data Movement. The capacity of off-chip memory is not sufficient to store larger-size problems, and thus adding more devices (e.g., CPUs or GPUs) is a viable option. The problem is now distributed over multiple devices and each device needs to exchange data through the PCIe or NVLink bus.

2.5.3 Inter-Device, Inter-Node Data Movement. Eventually, a single compute node may not be sufficient to support the number of devices required to perform the computations. Therefore, using multiple compute nodes becomes necessary. The network (e.g., InfiniBand) that connects the compute nodes is the weakest link out of the three, and thus it potentially limits the overall performance.

2.6 Related Work

Several groups studied acceleration of DG discretizations on GPUs. Chan et al. [4] used a single GPU to accelerate applications that use hybrid curvilinear meshes. Gandham et al. [12] used GPU to accelerate DG methods for solving shallow water equations. Karakus et al. [20] used GPU to accelerate DG methods for the unsteady incompressible Navier-Stokes equation. Mu et al. [30] used arbitrary, high-order DG (ADER-DG) for solving 3D elastic wave equation on GPUs, using unstructured tetrahedral meshes. They reached speed-up of 24.3x for single-precision and 12.8x for double-precision arithmetic on Nvidia Tesla C2075 GPU, compared to a single-core version of their CPU code that ran on Intel Xeon W5880. The work by Modave et al. [28] analyzed GPU performance for acoustic and elastic models, using a nodal DG method. Finally, the research by Hanindhito et. al. uses digital processing in-memory (PIM) to improve the performance of wave simulation based on DG discretization with GLL integration scheme [14]. Although the

[3] Communication cost is the key bottleneck in many industry-relevant applications.

result is promising, this emerging technology is not ready for deployment in HPC cluster in the near future. Furthermore, finding the optimized data layout to reduce inter-memory block data transfers is not automated at this point. None of the aforementioned works consider DG discretizations with GLL integration on straight-faced hexahedral elements on readily available multi-GPUs HPC cluster.

3 METHODS

3.1 Code Optimization and Verification

Table 1: Platform Hardware Configuration

Platform	PowerEdge R740	IBM AC922
CPU		
Model (# Sockets)	Intel Xeon 8160 (2)	IBM POWER9 (2)
Base/Turbo Clock	2.10GHz/3.70GHz	2.30GHz/3.80GHz
Total Cores/Threads	48/48	40/160
Inter-socket Link	UPI 41.6GBps	X-Bus 64GBps
GPU		
Model	NVIDIA Tesla V100	
# GPUs	2	4
Form Factor	PCIe FLFH	SXM2
Memory Type/Size	HBM2/16GB	HBM2/16GB
Memory Bandwidth	900GBps	900GBps
CPU-to-GPU Link	PCIe 3.0	NVLink 2.0
GPU-to-GPU Link	PCIe 3.0	NVLink 2.0
Computation Nodes		
# Nodes (# GPUs)	2 (4)	32 (128)
Inter-node Link	FDR MT27500	EDR MT28800

We have developed an optimized CPU-based code using DG discretization of the acoustic and elastic wave equations with hexahedral elements, along with tensor-product GLL quadrature (Section 4). We refer to this code as PDEBlaster. It relies on p4est [3] for mesh handling and MPI communication. Its accuracy has been verified by examining problems for which an analytical solution exists. Based on the CPU code, we have developed GAPS, the GPU-accelerated PDE solver for the corresponding wave simulation, with multiple flavors. We identify major bottlenecks of the code, and perform optimizations to improve performance (Section 6). Finally, we extend the scalability of GAPS to support multi-GPUs and multi-nodes for the simulation of industry-relevant problems (Section 7).

3.2 Environment and Platform

We use two different platforms for performance comparisons (Table 1). The Intel-based platform is mainly used for single-device code development, while the PowerPC-based platform is used for multi-GPU, multi-node developments, due to the availability of more GPUs per node, the number of nodes, and their connection via a high-bandwidth NVLink bus. [24, 25].

3.3 Performance Assessment

All codes developed in this work report diagnostics, such as time spent in each kernel, and L_2 error between a computed result, and the corresponding exact solution. For the purpose of performance assessment, all of the experiments run for 1000 time-steps.

Table 2: Wave Simulator Configurations

Problem Name	Flux Solver	Precision
Acoustic	Riemann	FP64
Elastic	Central	FP64
Elastic	Riemann	FP64
Acoustic	Riemann	FP32
Elastic	Central	FP32
Elastic	Riemann	FP32

Table 3: Configurations and Optimizations

Label	Hardware	MPI Library	Optimization
CPU_1	2 x POWER9	Spectrum MPI[1]	Basic
CPU_2	2 x Xeon 8160	Intel MPI[2]	Basic
GAPS_base	1 x Tesla V100	None	All Section 5
GAPS_fl	1 x Tesla V100	None	All Section 5, 6.1, 6.2
GAPS_fls	1 x Tesla V100	None	All Section 5, All Section 6

[1] We use IBM Spectrum MPI version 10.3.0.
[2] We use Intel MPI version 2018.2.199.

To distinguish between the effects of different optimization techniques, we label multiple versions of GAPS. PDEBlaster runs on multi-core processors using MPI (Table 3). GAPS_base is the parallel GPU code implementation described in Section 5. GAPS_fl is the optimized version of GPU code, which includes kernel fusion (Section 6.1) and LUT-based neighbor and node search (Section 6.2). Finally, GAPS_fls implements all optimizations described in Section 6, including improved utilization of the shared memory and SM-occupancy-aware register allocation.

There are six simulator configurations as shown in Table 2 that are used to assess performance both in single-GPU and multi-GPU runs. In single-GPU runs, the performance of each GPU optimization level is compared to the baseline multi-CPU runs (Section 8.1). We also compare the performance of two different mesh configurations (Section 8.3). In multi-GPU runs, the effect of ghost exchange optimization (Section 8.4) and the performance of MPI libraries (Section 8.5) are investigated. We also analyze the scalability of the GAPS multi-GPU implementation (Section 8.6).

3.4 GPU Kernel Profiling

To analyze the bottlenecks of our GPU implementation, we use nvprof [6] to characterize each GPU kernel. The profiling results help us identify the key bottlenecks, and perform necessary optimizations to improve performance. We also present the roofline chart [45] for each kernel (Section 8.2), where the NVIDIA Tesla V100 GPU's roofline model is obtained using the ERT [49].

4 CPU IMPLEMENTATION

4.1 Simulation Flow and Data Flow

PDEBlaster consists of two large loops as shown in Figure 2-ⓒ. The outer loop performs time-stepping, and is repeatedly executed at each time-step. The inner loop is the integration loop for the fourth-order, low-storage Runge-Kutta (LSRK4) integrator, which

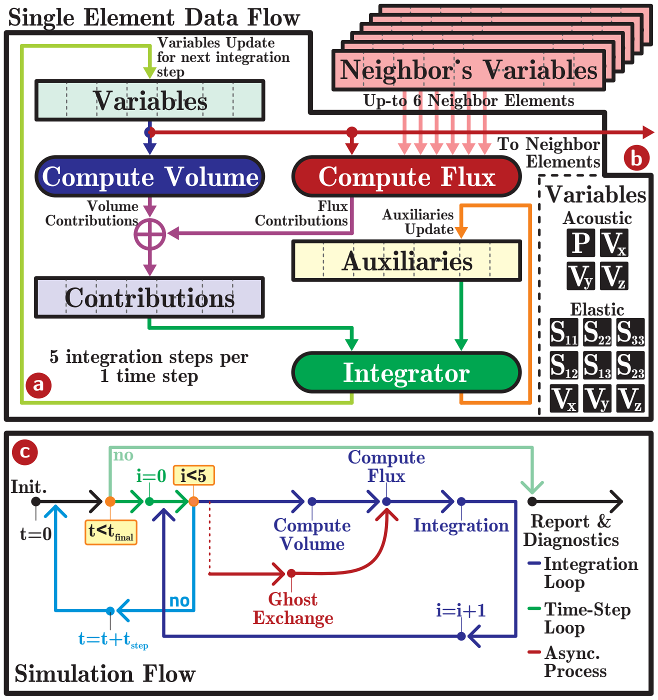

Figure 2: ⓐ The data flow of each element; ⓑ the `Variables` for both acoustic and elastic problems; ⓒ and the simulation flow.

has 5 steps [15]. Finally, the simulation kernels are called inside the integration steps, and are: `Volume`, `Flux`, and `Integration`. Therefore, each kernel is called five times per time-step.

There are 4 unknown variables for the acoustic, and 9 unknown variables for the elastic wave, respectively (Fig. 2-ⓑ). The `Volume` and `Flux` use `Variables` to compute the volume contributions and flux contributions, respectively (Fig. 2-ⓐ). Both volume and flux contributions are accumulated as `Contributions`, and then used by `Integration` to update the `Variables` for the next time-step. Moreover, `Integration` updates `Auxiliaries`, which is temporary storage for temporal integration using LSRK4 integrator.

4.2 Simulation Kernels

The acoustic and elastic wave simulators have different `Volume` and `Flux` kernels, although they both share the same `Integration` kernel, which operates on all nodes of an element (`Variables` and `Contributions`), and updates `Variables`.

`Volume` computations are local, and involve computing spatial derivatives of different quantities within an element. Using straight-faced hexahedral elements, along with GLL quadrature, decouples derivative calculations in each direction from other directions, and simplifies them into dot-products between a subset of `Variables` and a constant differential vector (Algorithm 1). These derivatives are then used to compute `Volume Contributions`.

We only have implemented the Riemann `Flux` for acoustic wave simulations, whereas, for the elastic case, we have implemented Central and Riemann `Fluxes`. Discussing the differences between different flux formulations is beyond the scope of this paper. Instead,

Algorithm 1: `Volume` kernel

```
Inputs: Variable vector of 3D tensors $\mathbf{u}^e$ and constant
  differential vectors $const\_dx, const\_dy, const\_dz$
Outputs: Contribution vector of tensors
for all $N^3$ nodes (i,j,k) within an element do
    for offset $o = 0, \ldots, N-1$ do
        for variable $r = 0, \ldots, R$ do some of
            $\frac{\partial u^e_{r,i,j,k}}{\partial x} \mathrel{+}= u^e_{r,(i+o)\%N,j,k} * const\_dx_o$
            $\frac{\partial u^e_{r,i,j,k}}{\partial y} \mathrel{+}= u^e_{r,i,(j+o)\%N,k} * const\_dy_o$
            $\frac{\partial u^e_{r,i,j,k}}{\partial z} \mathrel{+}= u^e_{r,i,j,(k+o)\%N} * const\_dz_o$
        end
    end
    $\forall r : c^e_{vol,r,i,j,k} = f(\nabla u^e_{0,i,j,k}, \ldots, \nabla u^e_{R,i,j,k})$
end
```

Algorithm 2: `Flux` kernel

```
Inputs: $\mathbf{u}^{e(f1)}, \mathbf{u}^{e'(f2)}$: 2D variable tensors of element $e$ and
  neighbor $e' \in \mathcal{N}(e)$ on faces $f1$ and $f2$
Outputs: Flux vectors of 2D tensors $\mathbf{c}_{\mathrm{f}}^{e(f1)}, \mathbf{c}_{\mathrm{f}}^{e(f2)'}$
for all $N^2$ face nodes (i,j) do
    for variable $r = 0, \ldots, R$ do
        $c^{e(f1)}_{f,r,i,j}, c^{e(f2)'}_{f,r,i,j} = g(u^{e(f1)}_{w,i,j}, u^{e'(f2)}_{w,i',j'}, w = 0, \ldots, R)$
    end
end
```

we focus on common computational patterns involved in different flux computations.

`Flux` computations occur at the nodes that reside on the shared interfaces between two elements, and require information from both sides (Algorithm 2), which then yield `Flux Contributions`.

4.3 Mesh Generation and Partitioning

`PDEBlaster` uses `p4est` to generate and partition nonconforming meshes. In this work, we focus only on uniform meshes, where all elements have the same size. The mesh is built at the beginning of a simulation and does not change throughout the simulation. The foundation that has been established in this work can be easily extended to support non-conforming meshes (Fig. 1-ⓙ).

The `p4est` manages the elements using a tree structure that allows iteration over elements using the provided `p4est_iterate` method. This method allows to iterate over each element for interior computation (needed for `Volume`), and to iterate over shared faces between neighboring elements (needed for `Flux`). The neighboring elements can be looked-up using pointer operations.

In `PDEBlaster`, each element is represented by a data structure called `ElementDataBase`. Inside, the values of each node in the element are stored in the `Variables`, `Contributions`, `Auxiliaries` and `Mass Inverse` arrays. We denote the total number of `Variables` as `NUM_VARIABLES` and the total number of nodes inside each element as `NNODE`, where `NNODE = NNODE_1D`^{3}. Therefore, the length

Table 4: ElementDataBase Data Structure Size

Problem	Precision	# Nodes	Size (Bytes)	
		(NNODE)	Standard	Ghost
Acoustic	FP32	512	32,832	8,200
Acoustic	FP64	512	65,640	16,400
Elastic	FP32	512	63,556	18,444
Elastic	FP64	512	127,088	36,888

of each array can be calculated as NUM_VARIABLES * NNODE. The standard size of each element is given in Table 4.

The total number of elements in PDEBlaster can be controlled by adjusting the refinement level. With higher refinement level, the problem domain is discretized into higher number of elements, where each element has smaller physical size to better represent the physical features. We denote the total number of elements as NUM_ELEMENT.

4.4 Ghost Exchange

For multi-CPU simulations, p4est creates load-balanced partitions of the mesh, and distributes them on multiple CPUs (Fig. 1-ⓗ). The p4est uses MPI to perform inter-CPU communications (e.g., during the computation of Flux, which needs neighboring elements' data). The neighboring element that is located on a different processor core is called ghost element, and MPI communication must be performed to exchange the ghost element data. This communication is called Ghost Exchange.

To perform a Ghost Exchange, p4est maintains two data structures called Ghost Layer and Ghost Buffer on each processor (Fig. 1-ⓘ). Ghost Layer is used to receive and store data from all ghost elements during a Ghost Exchange from neighboring processors. During the look-up operation for an element's neighbor, the pointer will automatically point to the Ghost Layer if the neighboring element lives in another processor. Ghost Buffer is used to store the local elements' data that will become ghost elements for a neighboring processor. The Ghost Buffer is updated just before Ghost Exchange. To overlap communication and computation, Ghost Exchange is performed asynchronously during the computation of Volume. (Fig. 2-ⓒ).

5 BASIC SINGLE-GPU IMPLEMENTATION

5.1 Basic Implementation

The development of the GPU code for GAPS in this section generally follows the CPU code (PDEBlaster) with some adjustments. GAPS tries to extract as much parallelism as possible to keep the Streaming Multiprocessors (SMs) of the GPU busy. Simulation results of GAPS are compared against PDEBlaster to ensure the GPU-accelerated version of the simulator retains the same numerical accuracy as the CPU counterpart.

The basic GPU code for GAPS implements each simulation kernel as a single CUDA kernel, and can only utilize a single GPU, which limits the size of the problem. We use p4est for mesh generation on CPU, copy the mesh data onto GPU memory, and then use our own developed tools for mesh handling on GPU. Then, the inner-loop and the outer-loop are executed as usual, except now they launch GPU kernels. At the end of the simulation, the results are copied back to the CPU memory.

Different opportunities for parallelization exist for each kernel, as described below.

5.1.1 *Volume kernel.*

The Volume computation, a completely local operation, is the most compute-intensive kernel of the simulation. It is highly parallelizable since each node within an element can be computed independently, and each element can be processed in-parallel. Thus, we extract the parallelism at element-level and node-level, with potential parallelism of NUM_ELEMENT * NNODE.

5.1.2 *Flux kernel.*

The Flux computation is a less compute-intensive kernel, since it is non-local, and requires neighbor information. In 3D space with NNODE=512 and NNODE_1D=8, Flux performs computations on all 6 faces of an element, each with 64 nodes. However, due to branches [47], this kernel is the most inefficient kernel for GPU. The branches are used to determine the neighboring elements. We need to serialize each face computation due to data hazard for nodes that touch more than one neighboring node (i.e., on the edge, or at the corner of the element). Moreover, the flux contributions for a pair of neighboring elements must be computed exactly once, and thus, we need a flag mechanism to avoid double computation. Therefore, potential parallelism will be less than NUM_ELEMENT * NNODE_1D * NNODE_1D.

5.1.3 *Integration kernel.*

The Integration computation is the smallest kernel, where memory accesses dominate kernel behavior. The potential parallelism that can be extracted is NUM_ELEMENT * NNODE * NUM_VARIABLES.

5.2 Data Structure

The tree-based data structure used by p4est is not suitable for GPUs, since pointer operations are too slow on GPUs, which hurts performance significantly. Therefore, we replace the tree-based data structure with an array, which is GPU-friendly, and aids obtaining the highest possible parallelism. Accordingly, all elements are stored in an array of ElementDataBase inside GPU memory.

5.3 Neighbor Look-up and Node Look-up

Because of moving from a tree-based data structure to an array-based data structure, we lose the mechanism of inferring neighbor elements by using pointer operations. In fact, traversing the mesh structure stored in linked-list is not GPU-friendly. Thus, we developed a method that allows inferring neighbor elements, which consists of simple loop calculations. This algorithm only works for conforming meshes, and thus we develop a LUT-based neighbor search, which will be discussed later. Moreover, we have developed algorithms to efficiently find corresponding nodes on an element's face, which is needed for Flux computations.

5.4 Judicious Pre-computations

On modern architectures, it is often more efficient to re-compute, than to pre-compute, store, and re-use [31]. However, pre-computing certain small-in-size, but frequently-used variables may improve efficiency. This strategy has been used in parts where it improved performance. For example, derivatives of shape functions are needed repeatedly, each time Volume is evaluated. We pre-compute the derivatives of shape functions. With NNODE=512, only 64 double- or single-precision constants were needed to be stored inside constant (symbol) memory of GPU, which is as fast as the register.

5.5 Loop Refactoring

Another basic optimization is loop refactoring, which reduces the number of branching operations that cause degradation in GPU performance. Loop refactoring includes flattening nested loops and merging independent loops. If a loop is short and has a constant number of iterations, we unroll the loop to eliminate the branching operations.

6 GPU OPTIMIZATIONS

The basic GPU code in Section 5 contains many optimizations, such as loop refactoring and judicious precomputations. However, we introduce additional GPU optimizations in this section.

6.1 Kernel Fusion

The basic GAPS implementation in Section 5 yields satisfactory performance improvements over the baseline CPU code of Section 4. However, it does not lead to peak GPU performance. Upon profiling the GPU code developed for GAPS in Section 5, the overall wave simulator is memory-bounded, even with 900 GBps of HBM2 memory bandwidth of the NVIDIA Tesla V100. Therefore, the next set of optimizations aim to reduce data movement.

Kernel fusion [42, 43] tries to merge two or more GPU kernels into one. Single kernel launch likely preserves the needed data near the SMs throughout kernel execution. This eliminates fetching data from global memory each time a small kernel is launched. A possible candidate for kernel fusion is merging Volume and Flux. The Integration must be kept as a separate kernel, since it updates the Variables, causing a data hazard. We want to update the Variables after the computations for all Contributions are completed. Thus, we do implicit grid-level synchronization using a separate kernel launch.

Multi-GPU and multi-node implementations, which will be discussed in the next section, require Ghost Exchange. To support the Ghost Exchange while Volume and Flux are fused, small modifications in Flux are needed. We split the Flux computation into Internal Flux and External Flux. The former indicates situations where neighbor elements are located on the same processor or GPU, whereas the latter indicates circumstances when a neighboring element is located on a different GPU.

Internal Flux can be computed based on data already available on a GPU, whereas External Flux must wait for Ghost Exchange to complete. Ghost Exchange will be overlapped with the computation of Volume and Internal Flux to hide inter-GPU communication overhead. With the fusion of these two kernels, one thread will handle one node throughout the kernel execution, and thus the potential parallelism that can be achieved is NUM_ELEMENT * NNODE.

6.2 Look-Up-Table-based Neighbor and Node Search

6.2.1 LUT-based Neighbor Search. Our implementation of the neighbor look-up algorithm on Section 5.3 is sub-optimal, since it involves a non-deterministic loop to determine the neighboring element. Moreover, this implementation does not support non-conforming meshes, where elements can have a different number of neighbors per face. Therefore, we propose a LUT-based neighbor search. The look-up table is implemented using two arrays inside GPU memory as shown in Figure 3. Both arrays are NUM_ELEMENT * NUM_CHILD * NUM_FACE long. The NUM_CHILD determines the number of neighboring elements on each face of an element. For a conforming mesh, NUM_CHILD is always equal to 1. NUM_FACE is the number of faces of each element. In 3D space, the NUM_FACE is 6.

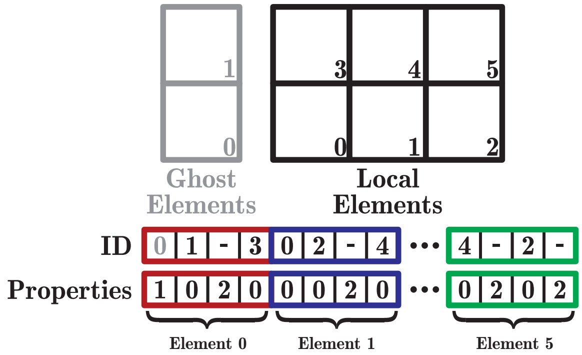

Figure 3: The look-up table structure used for neighbor search contains the ID and Properties arrays. The ID array stores indexes to the ElementDataBase array or GhostLayer, while the Properties array indicates whether the neighbor is a local element (0), a ghost element (1), or does not exist (2).

The first array stores the neighbor elements identification (ID) using 32-bit integers, which is basically the index to the array of ElementDataBase (or Ghost Layer, if it is a ghost element), stored in GPU memory. The second array stores the properties of the neighbor elements. The properties are stored using 8-bit integers, and indicate whether the neighbor lives on the same processor (local element), on a different processor (ghost element), or does not exist (i.e., for element located in boundary).

During simulation initialization, we construct the look-up table by calling p4est_iterate on the CPU. Then, the look-up table is copied to the GPU memory. Since our mesh structure remains identical throughout the simulation, the LUT only needs to be constructed once.

6.2.2 LUT-based Node Search. We also implement a LUT-based node search to replace our algorithm in Section 5.3. For NNODE=512, we need an array of 512, 8-bit integers. The 6 lower bits are used to identify in which side of the element a particular node is located at, while the upper 2 bits are unused. If a node is located in the interior of the element, then the bit value should be 0b00000000. If a node is located at the corner between face 0, face 2, and face 4, then the bit value should be 0b00010101. Without this LUT, we find the face nodes using multiple conditional branches, which is not GPU-friendly since it will introduce thread divergence.

6.3 Shared Memory and Register Allocation

6.3.1 Improved Shared Memory Utilization. Shared memory is a fast, user-managed, small, on-chip memory, which is available for each streaming multiprocessor (SM) of a GPU, and is used to share data between threads inside a thread block [46]. The shared memory is allocated when the kernel starts and de-allocated when the kernel completes. Starting with the Volta architecture, both the L1 cache and the shared memory are fused together; if the shared memory is not used, all of this unified memory is used as hardware-managed L1 cache. In other words, the use of the shared memory will reduce the L1 cache capacity and, thus, choosing the right data to be stored in the shared memory is crucial for performance.

There are two data candidates to be stored in shared memory. First, the `Contributions`, which are written repeatedly during `Flux` and `Volume`. Second, the `Variables`, which are read repeatedly during `Flux` and `Volume`. Storing both `Contributions` and `Variables` may not be the best idea for two reasons. First, as more shared memory is consumed, the capacity of L1 cache becomes smaller, which may degrade performance. Second, due to the nature of the memory accesses in `Volume`, storing `Variables` inside the shared memory complicates the memory accesses, potentially leading to no performance gain, especially when all nodes inside an element cannot be processed under the same thread block. Therefore, we only store `Contributions` inside shared-memory.

6.3.2 SM-Occupancy-Aware Register Allocation. Another concern is SM occupancy, which is determined by the number of thread blocks that can be scheduled on each SM. If there are enough blocks to occupy the SM, the scheduler will aggressively switch to another thread block and schedule it to run, while waiting for the previous block to finish its memory accesses, in order to hide the latency. We can instruct `nvcc` to allocate registers accordingly, enabling the SM to schedule more than one thread block using decorator `__launch_bounds__`. There may be some register spills, which means that the local memory is used due to insufficient registers [37]. A few register spills may not hurt performance, but we refactor the code to reduce the spills.

7 MULTI-GPU MULTI-NODE IMPLEMENTATION

7.1 Multi-GPU/Multi-Node Approach

While `p4est` (Section 4.3) is powerful for generation, partitioning, and handling of non-conforming meshes on CPUs, it does not support GPUs. In this section, we outline our strategy for adapting `p4est` for our multi-GPU, multi-node implementation: we generate and partition the mesh on the CPU, and then copy it to appropriate data structures on the GPU, thus improving workloads described in Sections 5.2 and 5.3.

We assign an MPI process to each GPU. The local rank identity for each MPI process is used to index the GPU. Therefore, the number of MPI ranks on a compute node is equal to the number of GPU cards living on that node. Once an MPI process is created and able to initialize its corresponding GPU, the flow will be similar to that in Section 5 and 6, with the addition of `Ghost Exchange` between GPUs.

7.2 CUDA-Aware MPI

CUDA-Aware MPI allows MPI to send and receive memory buffers instantiated in GPU memory. Without CUDA-aware MPI, the GPU memory buffer must be staged first on CPU memory using `cudaMemcpy`, before sending and receiving data through MPI communication, which is inefficient. Another feature that CUDA-Aware MPI supports is NVIDIA GPUDirect technologies for high-bandwidth, low-latency communications between NVIDIA GPUs, both intra- and inter-node. GPUDirect P2P allows the GPU-instantiated memory buffers to be exchanged directly between GPUs inside the same node via the fastest bus available between them (e.g., NVLink) while GPUDirect RDMA allows the GPU-instantiated memory buffers to be sent and received directly through a network adapter (e.g., InfiniBand NIC) without staging through CPU memory [24, 25].

Table 5: Comparison of MPI Implementations

MPI Name	Developer	CUDA-Aware	Async. Progress
MVAPICH[1] [32]	Ohio State Univ.	Yes	Yes
Spectrum MPI	IBM	Yes	Yes
Intel MPI	Intel	No	Yes

[1] We use MVAPICH2-GDR version 2.3.4.

7.3 Ghost Layer and Ghost Buffer Implementation

We briefly discussed `Ghost Exchange` for CPUs in Section 4.4. In Multi-GPU implementation, we move both `Ghost Layer` and `Ghost Buffer` arrays to the GPU's memory. Then, by utilizing GPUDirect RDMA in CUDA-aware MPI, the MPI send and receive for these arrays can be instantiated directly in GPU memory, with no CPU-GPU communication overhead.

We develop a GPU kernel to update `Ghost Buffer` contents before `Ghost Exchange` is initiated, since it is no longer updated by `p4est`. This kernel uses our efficient scheme to map between `Ghost Buffer` array and `ElementDataBase` array, since not all local elements are ghost elements for neighboring processors. Moreover, `Ghost Layer` does not need special kernel treatments, since GPUDirect RDMA stores the ghost elements received after `Ghost Exchange`, directly inside GPU memory. The LUT-based neighbor search discussed in Section 6.2 will automatically index the `Ghost Layer` array, if the neighboring element is a ghost element.

7.4 Ghost Exchange Optimization

During `Ghost Exchange`, ghost elements are still represented using the `ElementDataBase` data structure, as discussed in Section 4.3, which is inefficient for communication. Instead of sending the whole `ElementDataBase`, we only send data that is needed by the neighbor processor. These are the `Variables` and the `Materials`. Therefore, we create a new data structure, called `ElementDataBaseGhost`. On average, this ghost data structure has only 27% of the size of a standard `ElementDataBase` (Table 4).

7.5 Asynchronous Progress Support in MPI

In MPI standard [23, 27], the non-blocking semantics do not guarantee asynchronous progression when `MPI_Isend` and `MPI_Irecv` are being used. The actual data transfer is usually delayed until `MPI_Wait` is called [38, 40], and thus some parts of communication may not be overlapped with computation.

Some MPI implementations support asynchronous progression if explicitly enabled [17]. This should minimize the communication overhead during Ghost Exchange, thus improving overall performance in multi-GPU Multi-Node runs. Table 5 lists MPI libraries in our cluster that support asynchronous progression. We explore MVAPICH-GDR and IBM Spectrum MPI in our work as they are both CUDA-Aware, and support asynchronous progression.

8 EVALUATION

8.1 Performance Analysis

PDEBlaster uses multiple CPU cores to run wave simulations (e.g., 40 cores for CPU_1, and 48 cores for CPU_2). This leads to communication overhead between CPU cores during Ghost Exchange. On average, this overhead accounts for 9.5%, and 17% of total simulation time, for CPU_1 and CPU_2, respectively. The MPI library has an important role in communication performance, and thus, obtaining a suitable configuration is important.

Although both Intel MPI and Spectrum MPI have asynchronous progress support, using it does not improve the performance of our CPU runs. On the contrary, it leads to overall performance degradation: on average, 9.3% and 73.5% longer simulation times for CPU_1 and CPU_2, respectively. This degradation is due to oversubscription[4]: the number of MPI threads that are running (i.e., worker threads, plus asynchronous progression threads) are more than the available physical cores on the system. In our case, decreasing the number of worker threads would not help either since our simulator prefers to have more processing cores to be able to extract the parallelism. The CPU_2 has an average speed-up of 1.36x over CPU_1 for a problem with 32,768 elements.

Figure 4-ⓐ illustrates the speed-up of different flavors of the GAPS over PDEBlaster. For a problem with 32,768 elements, the average speed-up over CPU_1 is 45.67x, 69.30x, and 84.15x for GAPS_base, GAPS_fl, and GAPS_fls, respectively. The kernel fusion, LUT-based neighbor search and node look-up provide significant improvements across all simulator configurations, with an average speed-up of 1.74x over GAPS_base. Using shared memory and SM-occupancy-aware register allocation further improves performance, although each simulator configuration behaves differently. On average, these optimizations improve simulation speed by 1.17x over GAPS_fl; this is better than what NVIDIA claimed in [5], where combining L1 cache and shared memory can achieve 93% performance compared to when shared memory is used. The acoustic problems enjoy the benefit of these optimizations with up-to 1.5x speed-up over GAPS_fl, while the elastic problem with a Riemann flux solver sees the least performance improvement.

The acoustic problem has relatively short kernels, with less intermediate results, and thus, reduces the register spills, and allows multiple thread blocks to be scheduled in each SM to hide memory access latency. The elastic problem with Riemann flux solver has the longest kernels, with lots of intermediate results, which leads to more register spills, decreasing the number of thread blocks that can be scheduled by SMs for latency hiding.

Figure 4-ⓑ provides detailed insights about the time spent in each kernel during a simulation. For GAPS_base, the simulation spent most of the time in computing Volume, due to its higher arithmetic operations, followed by Flux, due to its execution inefficiencies caused by branches. In GAPS_fl and GAPS_fls, both Volume and Flux are fused into one kernel, which takes significantly less time compared to Volume plus Flux in GAPS_base.

8.2 GPU Kernel Profiling

Figure 4-ⓒ shows the GPU kernel characteristics in both FP64 and FP32 runs. The Integration kernel is clearly a memory-bound kernel. We ignored this kernel during our optimizations, since it is very short and it has little effect on the overall simulation performance. The Flux kernel also has a low execution efficiency, due to thread divergence, which is evident by the kernel being on the left side of the roofline plot, both for FP32 and FP64 runs. The Volume kernel is mostly located on the right side of the roofline plot; while it is still memory-bound, the arithmetic intensity of Volume is significantly higher than other kernels.

Fusing Volume and Flux moves the combined kernel characteristics to the top and right of their original position in the roofline model. This indicates the fused kernel has more GFLOPs/sec and more FLOPs/byte, which translates to an average of 1.74x speed-up. Finally, using shared memory moves the fused kernel to the top (increases GFLOPs/sec), albeit, slightly to the left of their original position, due to reduced L1-cache capacity. This translates to an average of 1.17x speed-up over the fused kernels without shared memory.

8.3 Effects of Element Order on Performance

High-order discretizations (i.e., elements with a larger number of nodes) are favored in many wave simulators, due to their ability to limit the dispersion error. We investigate whether high-order elements also have advantages from a hardware perspective. We use NNODE=64 and NNODE=512, which are labeled as N64 and N512. We keep the aggregate number of nodes the same, at approximately 63 Million for both cases, and run the simulation using GAPS_fls optimization for 1000 time-steps.

Figure 5 shows N512 is on average 1.12x faster than N64. The N512 case has 216 out of its 512 nodes assigned purely to Volume computations, which has high arithmetic intensity. Only 8 out of the 64 nodes in N64 perform purely Volume computations, meaning N64 needs to do more Flux computations for the same aggregate number of nodes, causing more inefficiencies due to the Flux kernel. Therefore, having high-order elements is also preferable from a hardware perspective. Figure 5 shows performance of each of these cases in a roofline model.

8.4 Ghost Exchange Overhead

By reducing the size of ElementDataBaseGhost to 27% of ElementDataBase, we are able to reduce the inter-GPU communication overhead inside the same node by as much as 72.81% (Fig. 6). The Baseline and Optimized use ElementDataBase and ElementDataBaseGhost for Ghost Exchange, respectively. The

[4] When asynchronous progress is enabled, both Intel MPI and Spectrum MPI spawn additional helper threads for each MPI process to handle communication. It is called "oversubscription" even though such threads are required for asynchronous progress. Such extra threads incur OS overhead, which can negate benefits and lead to performance degradation due to the reduced computing resources or overhead of thread context switching and core contentions. Note that GPU runs do not suffer from this problem when asynchronous progress is enabled since all computation is offloaded to the GPU and is thus unaffected by asynchronous communication handled by the CPU.

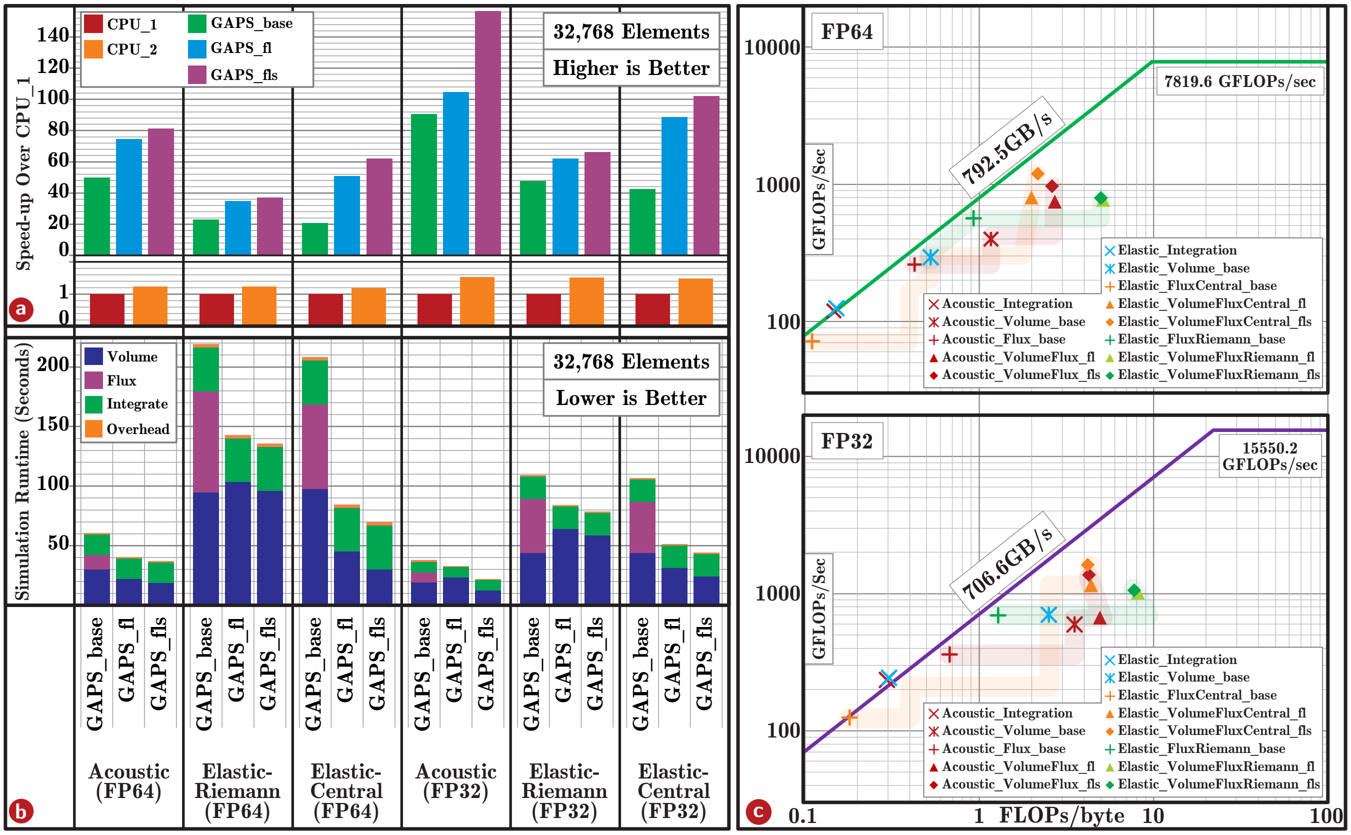

Figure 4: PDEBlaster compared against GPU simulations for 32,768 elements and 1000 time-steps. ⓐ Columns show speed-up over CPU_1, where higher is better; ⓑ time consumed by each GPU kernel, improvements due to optimizations, where lower is better; ⓒ Each GPU kernel can be characterized via the roofline model.

benefits of this optimization become apparent for multi-node runs that uses Infiniband as the communication link between nodes, where the communication overhead is reduced as much as 75.83%.

8.5 MPI Asynchronous Progress Impacts

We compare performance of different MPI libraries (Table 5) for 1 node with 4 GPUs (1N4G), and 4 nodes with 16 GPUs (4N16G) using the optimization described in Section 7.4 to reduce communication overhead. The MVAPICH2-GDR does not have asynchronous progress explicitly enabled while both MVAPICH2-GDR_async and SpectrumMPI_async have asynchronous progress explicitly enabled. Results are shown in Fig. 7.

With asynchronous progress enabled (MVAPICH2-GDR_async), the communication overhead is reduced by an average of 56.74% for both 1N4G and 4N16G, compared to MVAPICH2-GDR. On POWERPC-based platform, SpectrumMPI_async has an average of 51.27% less communication overhead on 1N4G, when compared to MVAPICH2-GDR_async, for obvious reasons, but falls behind MVAPICH2-GDR_async in 4N16G with 33% more communication overhead, due to the InfiniBand multi-rail utilization inefficiency. Overall, the asynchronous progress improves the ability to overlap the Ghost Exchange with Volume and Internal Flux computation, and thus hiding the communication overhead.

8.6 Multi-GPU Multi-Node Scalability

Our multi-GPU and multi-node implementation of GAPS achieves weak-scaling over 128 GPUs on 32 compute nodes. Figure 8 shows the scalability of our simulator, using SpectrumMPI_async. Due to limited space, we only show elastic simulations with a central flux solver, for FP32 and FP64. However, other cases enjoy the same characteristics. The four GPUs run (i.e., single node) enjoys the benefit of high-bandwidth NVLink. When multiple nodes are used, InfiniBand becomes the weakest link, which limits performance.

9 FUTURE WORK

There are still room to be explored for further improving our wave-simulation performance. Reducing the data movement overhead, both intra-device and inter-device, will be the focus of our future research.

To reduce intra-device data movement overhead that limits single-GPU performance, better shared-memory utilization is still our main focus, especially with newer GPU that has even larger shared-memory size. Furthermore, exploring the extension of the CUDA programming model called Cooperative Groups allow us to organize threads at a granularity that matches with the applications (i.e., at element- or node-level). This specifically useful for synchronization of threads during Flux computation to avoid race

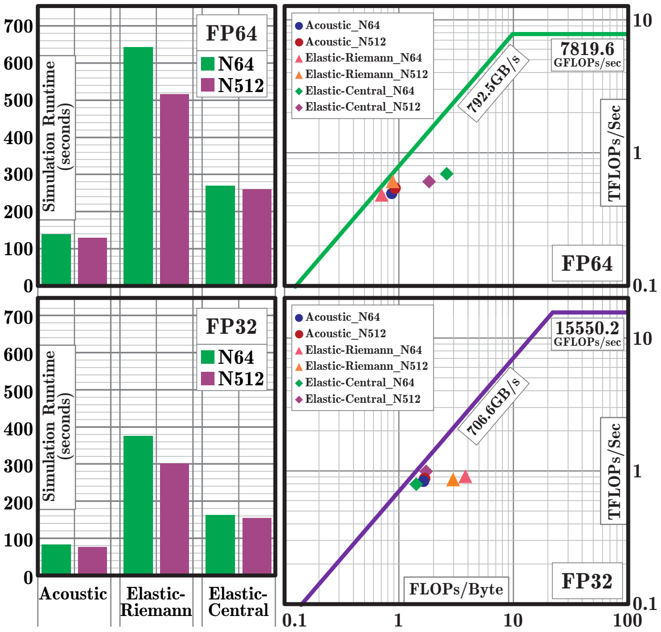

Figure 5: Simulation performance and roofline model of GAPS_fls under different mesh configurations for NNODE=64 and NNODE=512 in FP64 (top) and FP32 (bottom) simulation runs.

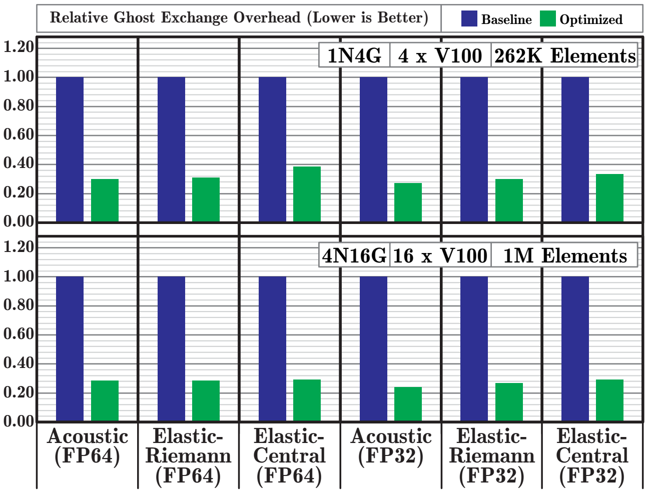

Figure 6: Ghost exchange overhead before and after optimization described in Section 7.4.

conditions. Furthermore, we plan to explore whether it is possible to fuse the Integration kernel to improve locality.

On the other hand, reducing the overhead of inter-device data movement is important for multi-GPU scalability. While weak-scaling is achieved in our current work, exploring strategies to reduce the communication overhead even further will improve the overall simulation performance. For example, sending only the nodes data that are located on the face of elements can reduce the communication overhead even more, compared to the optimized

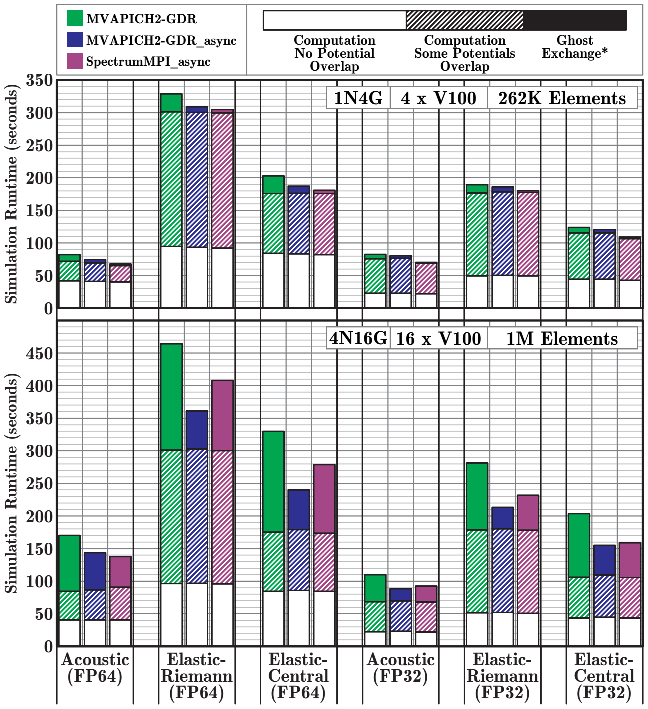

Figure 7: Ghost exchange overhead in single-node multi-GPU (top) and multi-node multi-GPU (bottom) runs under different MPI libraries with or without asynchronous progression explicitly enabled. The solid color indicates ghost exchange overhead, while the stripe color and white color indicate time spent in computation that can (volume and internal flux) or cannot (external flux and integration) be overlapped, respectively.

Ghost Exchange presented in Figure 6. This modification requires slightly more complicated indexing to access the data, and thus additional computation. Furthermore, the use of mixed-precision, where the lower precision is used for communication to reduce the communication overhead while higher precision is used for computation to maintain numerical accuracy and stability, is also promising.

10 CONCLUSIONS

We explored the performance of a class of wave simulations that use DG finite element method, along with GLL integration scheme on hexahedral elements with straight faces. These simulations have lower arithmetic intensity compared to simulations that use general meshes. This makes attaining peak performance of GPUs challenging, primarily due to excessive data movement between off-chip and on-chip memory, and larger communication to computation ratio in multi-GPU configurations. Algorithms with these properties are pervasive in industrial applications.

We deployed several optimization strategies for GPU kernels, yielding satisfactory improvements across various wave-simulator configurations. Kernel fusion and LUT-based neighbor search optimizations yield up to 2.6x speed-up over the basic GPU implementation, while improved shared memory usage and SM-occupancy-aware register allocation provided an additional 1.49x speed-up.

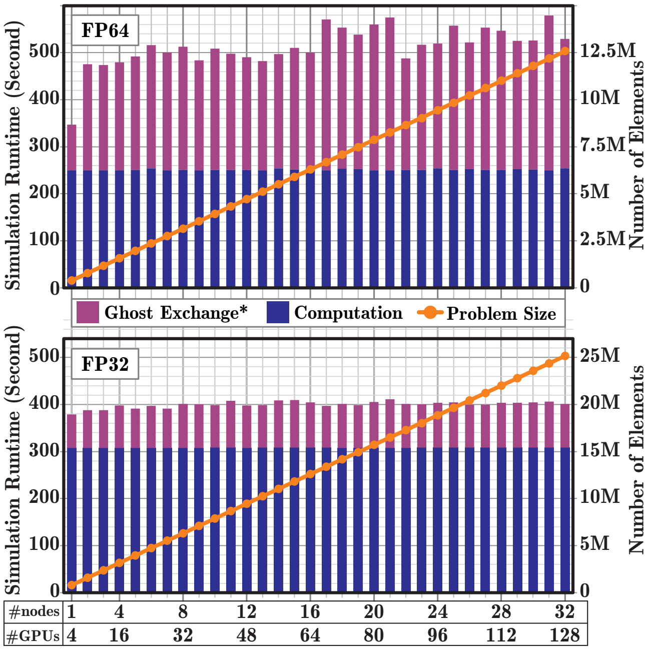

Figure 8: The multi-GPU multi-node scalability on the elastic problem with a central flux solver for FP32 run (top) and FP64 run (bottom). The problem size distributed to each GPU is kept the same as we scale to more GPUs.

We considered strategies to reduce the intra- and inter-node communication overhead for large-scale simulations on multi-GPU, multi-node configurations. These include: a) reducing the data size that needs to be exchanged, leading to (on average) 70.27% less communication time; and b) using an MPI implementation that supports GPUDirect RDMA, GPUDirect P2P, and asynchronous progression, which further reduces the overhead by 82.03%. Finally, our implementation achieves weak-scaling over 128 GPUs.

ACKNOWLEDGMENTS

This research was supported in part by ExxonMobil Technology and Engineering, agreement no. EM10480.36, National Science Foundation (NSF) grant number 1763848, and computational resources from Texas Advanced Computing Center (TACC) with allocation number ASC20005, DMS21075, and A-ee6. Any opinions, findings, conclusions, or recommendations are those of the authors and not of the sponsors.

REFERENCES

[1] Daniel S Abdi, Lucas C Wilcox, Timothy C Warburton, and Francis X Giraldo. 2019. A GPU-accelerated continuous and discontinuous Galerkin non-hydrostatic atmospheric model. *The International Journal of High Performance Computing Applications* 33, 1 (2019), 81–109.

[2] Aria Abubakar, Gong Li Wang, Lin Liang, Tarek M. Habashy, and Maokun Li. 2016. Electromagnetic modeling and inversion application for oil and gas industry. In *2016 Progress in Electromagnetic Research Symposium (PIERS)*. 938–938. https://doi.org/10.1109/PIERS.2016.7734532

[3] Carsten Burstedde, Lucas C Wilcox, and Omar Ghattas. 2011. p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing* 33, 3 (2011), 1103–1133.

[4] Jesse Chan, Zheng Wang, Axel Modave, Jean Francois Remacle, and T. Warburton. 2016. GPU-accelerated discontinuous Galerkin methods on hybrid meshes. *J. Comput. Phys.* 318 (2016), 142–168. https://doi.org/10.1016/j.jcp.2016.04.003 arXiv:1507.02557

[5] NVIDIA Corporation. 2017. NVIDIA TESLA V100 GPU ARCHITECTURE: THE WORLD'S MOST ADVANCED DATA CENTER GPU. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[6] NVIDIA Corporation. 2019. NVIDIA CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/profiler-users-guide/index.html/.

[7] Aditya M. Deshpande and Jeffrey T. Draper. 2015. Modeling Data Movement in the Memory Hierarchy in HPC Systems. In *Proceedings of the 2015 International Symposium on Memory Systems* (Washington DC, DC, USA) *(MEMSYS '15)*. Association for Computing Machinery, New York, NY, USA, 158–161. https://doi.org/10.1145/2818950.2818972

[8] T. Duda, J. Bonnel, E. Coelho, and K. Heaney. 2019. Computational Acoustics in Oceanography: The Research Roles of Sound Field Simulations. *Acoustics Today* 15 (2019), 28–37. Issue 3.

[9] Arash Fathi, Loukas F Kallivokas, and Babak Poursartip. 2015. Full-waveform inversion in three-dimensional PML-truncated elastic media. *Computer Methods in Applied Mechanics and Engineering* 296 (2015), 39–72.

[10] Arash Fathi, Babak Poursartip, and Loukas F. Kallivokas. 2015. Time-domain hybrid formulations for wave simulations in three-dimensional PML-truncated heterogeneous media. *Internat. J. Numer. Methods Engrg.* 101, 3 (2015), 165–198.

[11] Arash Fathi, Babak Poursartip, Kenneth H. Stokoe II, and Loukas F. Kallivokas. 2016. Three-dimensional P- and S-wave velocity profiling of geotechnical sites using full-waveform inversion driven by field data. *Soil Dynamics and Earthquake Engineering* 87 (2016), 63 – 81.

[12] Rajesh Gandham, David Medina, and Timothy Warburton. 2015. GPU Accelerated Discontinuous Galerkin Methods for Shallow Water Equations. *Communications in Computational Physics* 18, 1 (2015), 37–64. https://doi.org/10.4208/cicp.070114.271114a arXiv:1403.1661

[13] Lluís Guasch, Oscar Calderon Agudo, Meng-Xing Tang, Parashkev Nachev, and Michael Warner. 2020. Full-waveform inversion imaging of the human brain. *npj Digital Medicine* 3 (2020), 1 – 12.

[14] Bagus Hanindhito, Ruihao Li, Dimitrios Gourounas, Arash Fathi, Karan Govil, Dimitar Trenev, Andreas Gerstlauer, and Lizy John. 2021. Wave-PIM: Accelerating Wave Simulation Using Processing-in-Memory. In *50th International Conference on Parallel Processing* (Lemont, IL, USA) *(ICPP 2021)*. Association for Computing Machinery, New York, NY, USA, Article 8, 11 pages.

[15] J.S. Hesthaven and T. Warburton. 2010. *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications*. Springer.

[16] S. B. Hong, N. Vlahopoulos, R. M. Mantey, and D. J. Gorsich. 2004. A computational approach for evaluating the probability of acoustic detection of a military vehicle. In *Targets and Backgrounds X: Characterization and Representation*, Wendell R. Watkins, Dieter Clement, and William R. Reynolds (Eds.), Vol. 5431. International Society for Optics and Photonics, SPIE, 150 – 159.

[17] Masashi Horikoshi, Balazs Gerofi, Yutaka Ishikawa, and Kengo Nakajima. 2022. Exploring Communication-Computation Overlap in Parallel Iterative Solvers on Manycore CPUs Using Asynchronous Progress Control. In *International Conference on High Performance Computing in Asia-Pacific Region Workshops* (Virtual Event, Japan) *(HPCAsia 2022 Workshop)*. Association for Computing Machinery, New York, NY, USA, 29–39. https://doi.org/10.1145/3503470.3503474

[18] Frank Ihlenburg. 1998. *Finite element analysis of acoustic scattering*. Springer.

[19] L.F. Kallivokas, A. Fathi, S. Kucukcoban, K.H. Stokoe, J. Bielak, and O. Ghattas. 2013. Site characterization using full waveform inversion. *Soil Dynamics and Earthquake Engineering* 47 (2013), 62 – 82.

[20] A. Karakus, N. Chalmers, K. Świrydowicz, and T. Warburton. 2019. A GPU accelerated discontinuous Galerkin incompressible flow solver. *J. Comput. Phys.* 390 (2019), 380–404. https://doi.org/10.1016/j.jcp.2019.04.010 arXiv:1801.00246

[21] J.E. Kozdon and L.C. Wilcox. 2018. An Energy Stable Approach for Discretizing Hyperbolic Equations with Nonconforming Discontinuous Galerkin Methods. *Journal of Scientific Computing* 76 (2018), 1742–1784.

[22] Martin-D Lacasse, Laurent White, Huseyin Denli, and Lingyun Qiu. 2018. Full-wavefield inversion: An extreme-scale PDE-constrained optimization problem. In *Frontiers in PDE-Constrained Optimization*. Springer, 205–255.

[23] Ignacio Laguna, Ryan Marshall, Kathryn Mohror, Martin Ruefenacht, Anthony Skjellum, and Nawrin Sultana. 2019. A Large-Scale Study of MPI Usage in Open-Source HPC Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 31, 14 pages. https://doi.org/10.1145/3295500.3356176

[24] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. 2020. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2020), 94–110. https://doi.org/10.1109/TPDS.2019.2928289

[25] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Xu Liu, Nathan Tallent, and Kevin Barker. 2018. Tartan: Evaluating Modern GPU Interconnect via a Multi-GPU Benchmark Suite. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. 191–202. https://doi.org/10.1109/IISWC.2018.8573483

[26] Elena Lucano, Micaela Liberti, Gonzalo G. Mendoza, Tom Lloyd, Maria Ida Iacono, Francesca Apollonio, Steve Wedan, Wolfgang Kainz, and Leonardo M. Angelone. 2016. Assessing the Electromagnetic Fields Generated By a Radiofrequency MRI Body Coil at 64 MHz: Defeaturing Versus Accuracy. *IEEE Transactions on Bio-Medical Engineering* 63, 8 (8 2016).

[27] Message Passing Interface Forum. 2015. *MPI: A Message-Passing Interface Standard Version 3.1.* https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

[28] A. Modave, A. St-Cyr, and T. Warburton. 2016. GPU performance analysis of a nodal discontinuous Galerkin method for acoustic and elastic models. *Computers and Geosciences* 91 (2016), 64–76. https://doi.org/10.1016/j.cageo.2016.03.008 arXiv:1602.07997

[29] Nazmul Haque Mondol. 2015. Seismic Exploration. In *Petroleum Geoscience*. Vol. 41. Springer Berlin Heidelberg, Berlin, Heidelberg, 427–454. https://doi.org/10.1007/978-3-642-34132-8_17

[30] Dawei Mu, Po Chen, and Liqiang Wang. 2013. Accelerating the discontinuous Galerkin method for seismic wave propagation simulations using the graphic processing unit (GPU)-single-GPU implementation. *Computers and Geosciences* 51 (2013), 282–292. https://doi.org/10.1016/j.cageo.2012.07.017

[31] Catherine Olschanowsky, Michelle Mills Strout, Stephen Guzik, John Loffeld, and Jeffrey Hittinger. 2014. A Study on Balancing Parallelism, Data Locality, and Recomputation in Existing PDE Solvers. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 793–804. https://doi.org/10.1109/SC.2014.70

[32] Dhabaleswar Kumar Panda, Hari Subramoni, Ching-Hsiang Chu, and Mohammadreza Bayatpour. 2021. The MVAPICH project: Transforming research into high-performance MPI library for HPC community. *Journal of Computational Science* 52 (2021), 101208. Case Studies in Translational Computer Science.

[33] Seymour V. Parter. 1999. On the Legendre–Gauss–Lobatto Points and Weights. *Journal of Scientific Computing* 14, 4 (1999), 347–355. https://doi.org/10.1023/a:1023204631825

[34] Babak Poursartip, Arash Fathi, and Loukas F. Kallivokas. 2017. Seismic wave amplification by topographic features: A parametric study. *Soil Dynamics and Earthquake Engineering* 92 (2017), 503–527.

[35] Babak Poursartip, Arash Fathi, and John L. Tassoulas. 2020. Large-scale simulation of seismic wave motion: A review. *Soil Dynamics and Earthquake Engineering* 129 (2020), 105909.

[36] A. Quarteroni and A. Valli. 1994. *Numerical Approximation of Partial Differential Equations*. Springer.

[37] Prashant Singh Rawat, Fabrice Rastello, Aravind Sukumaran-Rajam, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. 2018. Register Optimizations for Stencils on GPUs. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Vienna, Austria) *(PPoPP '18)*. Association for Computing Machinery, New York, NY, USA, 168–182. https://doi.org/10.1145/3178487.3178500

[38] Amit Ruhela, Hari Subramoni, Sourav Chakraborty, Mohammadreza Bayatpour, Pouya Kousha, and Dhabaleswar K. Panda. 2018. Efficient Asynchronous Communication Progress for MPI without Dedicated Resources. In *Proceedings of the 25th European MPI Users' Group Meeting* (Barcelona, Spain) *(EuroMPI'18)*. Association for Computing Machinery, New York, NY, USA, Article 14, 11 pages.

[39] R. L. Sengbush. 1983. *Seismic Exploration Methods*. Springer Netherlands, Dordrecht.

[40] Min Si and Pavan Balaji. 2017. Process-Based Asynchronous Progress Model for MPI Point-to-Point Communication. In *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 206–214. https://doi.org/10.1109/HPCC-SmartCity-DSS.2017.27

[41] Didem Unat, Anshu Dubey, Torsten Hoefler, John Shalf, Mark Abraham, Mauro Bianco, Bradford L. Chamberlain, Romain Cledat, H. Carter Edwards, Hal Finkel, Karl Fuerlinger, Frank Hannig, Emmanuel Jeannot, Amir Kamil, Jeff Keasler, Paul H J Kelly, Vitus Leung, Hatem Ltaief, Naoya Maruyama, Chris J. Newburn, and Miquel Pericás. 2017. Trends in Data Locality Abstractions for HPC Systems. *IEEE Transactions on Parallel and Distributed Systems* 28, 10 (2017), 3007–3020. https://doi.org/10.1109/TPDS.2017.2703149

[42] Mohamed Wahib and Naoya Maruyama. 2014. Scalable Kernel Fusion for Memory-Bound GPU Applications. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 191–202. https://doi.org/10.1109/SC.2014.21

[43] Jiajun Wang, Ahmed Khawaja, George Biros, Andreas Gerstlauer, and Lizy K. John. 2016. Optimizing GPGPU Kernel Summation for Performance and Energy Efficiency. In *2016 45th International Conference on Parallel Processing Workshops (ICPPW)*. 123–132. https://doi.org/10.1109/ICPPW.2016.32

[44] L.C. Wilcox, G. Stadler, C. Burstedde, and O. Ghattas. 2010. A high-order discontinuous Galerkin method for wave propagation through coupled elastic–acoustic media. *J. Comput. Phys.* 229, 24 (2010), 9373 – 9396.

[45] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (apr 2009), 65–76. https://doi.org/10.1145/1498765.1498785

[46] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. 2015. Enabling and Exploiting Flexible Task Assignment on GPU through SM-Centric Program Transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (Newport Beach, California, USA) *(ICS '15)*. Association for Computing Machinery, New York, NY, USA, 119–130. https://doi.org/10.1145/2751205.2751213

[47] Bo Wu, Zhijia Zhao, Eddy Zheng Zhang, Yunlian Jiang, and Xipeng Shen. 2013. Complexity Analysis and Algorithm Design for Reorganizing Data to Minimize Non-Coalesced Memory Accesses on GPU. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Shenzhen, China) *(PPoPP '13)*. Association for Computing Machinery, New York, NY, USA, 57–68. https://doi.org/10.1145/2442516.2442523

[48] Wm A Wulf and Sally A McKee. 1995. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news* 23, 1 (1995), 20–24.

[49] Charlene Yang. 2015. Berkeley CS Roofline Toolkit. https://bitbucket.org/berkeleylab/cs-roofline-toolkit.