Copyright
by
Bagus Hanindhito
2025

The Dissertation Committee for Bagus Hanindhito
certifies that this is the approved version of the following dissertation:

Algorithmic and Architectural Optimizations for Discontinuous Galerkin Finite Element Simulations

Committee:

Lizy Kurian John, Supervisor

Andreas Gerstlauer

Earl Swartzlander

George Biros

Arash Fathi

Algorithmic and Architectural Optimizations for Discontinuous Galerkin Finite Element Simulations

by

Bagus Hanindhito

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2025

Dedication

Dedicated to:

my parents Wijayanti and Djatmiko,

my wife Fitria Ridayanti,

my children Fathimah Imtiyaz and Fatih Imtiyaz,

and my homeland, The Republic of Indonesia

Epigraph

If all the trees on Earth were pens and the ocean 'were ink,'
refilled by seven other oceans, the Words of Allah would not be exhausted.
Surely Allah is Almighty, All-Wise.
—Qur'an, Luqman 31:27

Acknowledgments

As hyperbolic[1] as it gets, it takes a village to write a dissertation and get a Ph.D., and thus, I would like to dedicate a few pages to thanking the people who supported me during my Ph.D. journey, directly or indirectly.

First and foremost, I would like to express my deepest gratitude and special appreciation to my advisor, Prof. Lizy K. John, for shaping my Ph.D journey and making it the most memorable and impactful in my life. My research during my Ph.D. would not have been possible without her guidance, advice, and support. I learned and built strong foundations on workload characterization, performance evaluation/benchmark, and architecture-aware optimizations from her. I would also like to praise her for her kindness to my wife and daughter. Arranging a baby shower at the lab group meeting for my wife when she was pregnant with our first child was a memorable experience for me and my wife. Ph.D. is a very long journey and would not be as enjoyable without a supportive advisor like her. One of my favorite quotes from her, *"Every paper has a home; your job is to find it (i.e., conferences, journals)."* She later told me it was a quote from the late Dr. Jim Browne of the UT CS Department.

Secondly, I would like to express my sincere gratitude to my dissertation committee: Prof. Andreas Gerstlauer, Prof. Earl Swartzlander, Prof. George Biros, and Dr. Arash Fathi. I am pleased to collaborate with Andreas on the research, which has become part of my dissertation. I appreciate his thorough evaluation and detailed suggestions to improve my work. Earl has taught me valuable lessons on computer arithmetic, an essential puzzle to the performance of computer systems. Interacting with George helped broaden my views on scientific applications, their algorithms, and their interaction with the hardware, especially on GPUs and FPGAs. Arash is my

[1] Interestingly, the work on this dissertation focuses on solving hyperbolic PDEs; no pun intended.

committee member and close friend, with whom I share and chat about many things outside of research. I am delighted that I have crossed paths with him. I would also like to thank the department staff whose roles are critical but often overlooked, tirelessly handling administrative and financial matters and helping me to finish my dissertation: Barbara Heine, Barry Levitch, David Korts, Leticia Lira, Lisa Contes, Thomas Atchity, Melanie Gulick, and Melody Singleton.

Thirdly, I would like to thank my comrades from the Laboratory for Computer Architecture (LCA) research group, who made my Ph.D. journey as much fun as possible: Alan Bacellar, Allison Seigler, Aman Arora, Ashen Ekanayake, Harsh Gugale, Jiajun Wang, Molly O'Neil, Mugdha Jadao, Qinzhe Wu, Ruihao Li, Shagnik Pal, Shashank Nag, Shuang Song, Siyuan Ma, Snehil Verma, Steven Flolid, Zachary Susskind, and Zhigang Wei. I am grateful for our bonds and friendships and for spending time with me during the darkest and brightest moments. I will miss the weekly group meeting with Potato-Egg-Cheese tacos from Taco Joint, everyday chats and gossip, spending nights together during the paper submission deadline, and end-of-semester lunch at Prof. John's house.

Fourthly, I would also like to thank my friends from the System-Level Architecture and Modeling (SLAM) research group, which happens to be the neighboring lab of LCA: Alexander Cathis, Dimitrios Gourounas, Kamyar Mirzazad, Kishore Punniyamurthy, Matthew Barondeau, and Mochamad Asri. Special mention goes to Dimitrios, with whom I have worked on the same project for five years. He accelerates the wave simulations using FPGAs, while I handle acceleration using GPUs; basically, we have been competing with each other! I learned a lot from him about all the challenging aspects of big design like this, from doing design-space explorations, planning the architectures, writing custom Python scripts to generate Verilog files, and verifying the synthesized design. Moreover, I would like to mention Mochamad, the first Indonesian I met on the ACSES (Architecture, Computer Systems, Embedded Systems) track of the Electrical and Computer Engineering (ECE) Department. He convinced the graduate admission that Institut Teknologi Bandung (ITB), the

university where I have my B.S. degree, is the top university in Indonesia. Nobody in the Department would have known ITB if not for him, potentially reducing my chance of getting admission to the Department as a Ph.D. student. Other ECE students I would like to mention include Grant Guglielmo, Hyunsu Chae, Joseph Dean, Michael Shamoun, Reshma Rajarama Nayak, Shijia Wei, Shvetha Senthil Kumar, and Tosin Jemilehin, I have worked and interacted with them during my Ph.D., either as a team on class project or as friends outside research.

Next, I would like to thank Indonesian students who became my close friends during my stay in Austin: Achmad Zulfikar, Addian Mirsha, Adityo Andikaputra, Adry Gracio, Aldio Wahyu, Aya Bangun, Dewi Azizah, Dirga Muhammad, Eka Ratnasari, Fahran Kamili, Ismail Halim, Joseph Butarbutar, Josia Simanjuntak, Melianna Ulfah, Oktarico Susilatama, Rahma, Reynaldy Fifariz, Taufik Al Amin, and Vincentius Sebastian. Special mention goes to Achmad and Josia, who accompanied me in exploring different parts of Texas and the United States. I also spent two exciting weeks with them, visiting mainland China (Beijing, Chengdu, Shenzhen, Shanghai) and Hong Kong as part of the Global Management course offered by McCombs School of Management. I would also like to mention Reynaldy for helping my wife and me during our early years in Austin, inheriting the Honda CR-V, and providing countless household furniture and items to furnish our little apartment.

While I am far away from my homeland, with the great circle distance between Austin, TX, and my hometown, Bandung, Indonesia, of over 10,000 miles (16,000 km), I can still feel as if I am at home thanks to the friendly and supportive Indonesian communities in Austin and its surrounding cities. I want to mention Indonesian communities who make my life outside in Texas more vibrant with their hospitality: Pengajian Austin, San Antonio, dan Killen (PASAK/Austin Halaqah), Rumah Budaya Indonesia di Austin (Indonesian Cultural House in Austin), Indonesia Diaspora Network di Austin (IDN Austin Chapter), and Persatuan Mahasiswa Indonesia di Amerika Serikat cabang Austin (PERMIAS/Indonesian Students' Association in United States, Austin Chapter). I would also like to thank the Consulate General

of Indonesia in Houston for their assistance during my study in the U.S., either in renewing my passport, casting my vote, or holding cultural and food festivals.

Specifically, I would like to thank my fellow Indonesians in greater Austin area who have been supporting me and my family during my Ph.D.: Mas Randy, Mbak Qoni, Mas Besar, Mbak Tia, Mas Candra, Mbak Gina, Mas Wardana, Mbak Melita, Bapak Djoko, Ibu Icha, Bapak Widodo, Bu Airin, Bapak David, Bu Iin, Mas Emil, Mbak Vitria, Mas Andika, Mbak Kiki, Mas Cardiff, Mbak Ida, Mbak Septri, Mbak Desi, Mbak Karisha, Mbak Refika, Mbak Geni, Mbak Sandra, Mbak Febi, Mbak Dian, and others cannot be mentioned one by one. Regular interaction with them through lunches, dinners, picnics, potlucks, hiking, barbecues, and road trips ensured I had other life besides my Ph.D. life.

I am grateful for the support from my industry's collaborators for my Ph.D. research. I would like to thank ExxonMobil Technology and Engineering Company for the opportunities to collaborate to develop high-performance wave simulations running on modern hardware architectures, which become part of this dissertation. Especially, I would like to mention Laurent White, Dimitar Trenev, Karan Govil, and Grant Seastream who have contributed to the research. I would also like to thank Dell Technologies for supporting my research during my Ph.D. Special thanks to Ramesh Radhakrishnan, who enabled the research project collaboration in 2018 and introduced me to the fantastic team, the Chief Technology and Innovation Office (CTIO) at Dell Technologies, and the awesome manager, Stephen Rousset. As much as I enjoy collaborating with the CTIO team, I had five internships working on exciting projects on HPC and AI/ML. Kudos to my teammates at Dell: Bhavesh Patel, Daniel Amann, Dennis Norwood, James Singer, Mehrad Yasrebi, Kevin Marks, Khushboo Rathi, Liz Raymond, and Sarah Griffin.

In addition, I want to thank NVIDIA for its Applied Research Accelerator Program, which provided me with DGX-A100, a fantastic platform on which I run my experiments. Moreover, I appreciate the Texas Advanced Computing Center

(TACC) for providing me access to leadership high-performance computing clusters to perform my research, including Stampede2, Maverick2, Longhorn, Frontera, and Lonestar6. I would also like to thank the Semiconductor Research Corporation (SRC) and the National Science Foundation (NSF) for supporting my Ph.D. research.

My Ph.D. journey at UT Austin is incomplete without mentioning my undergraduate professor, Yudi Satria Gondokaryono. Although he is no longer with us, he inspired me to be a computer engineer. He suggested I apply for a Ph.D. at UT Austin, his dream school, where he wanted to pursue his Ph.D. but ended up at another university. I still remember that day when he talked a lot about Austin, which, in his opinion, is an excellent city for doing a Ph.D., working at tech companies, and raising kids. Following his advice, I applied to UT Austin, and he wrote me a recommendation letter. Three days after he submitted the letter, he passed away. Almost seven years later, I am glad I followed his suggestion and confirmed his correct opinion about Austin! He would have been pleased to know that I had been admitted to UT Austin four months after he passed away and graduated with my Ph.D.

To wrap up, I keep the most special for the last; that is my family. I want to thank my wife, Fitria Ridayanti, for her patience while I was pursuing my Ph.D. for over six years. She always gives me positive energy whenever I have a problem with my research, encourages me to look at the positive sides, and keeps me in her prayers. I would like to express my deepest gratitude to my parents, Wijayanti and Djatmiko, and my parents-in-law, Yunidar and Marsidi, who tirelessly keep me in their prayers even though we are far apart. Shout out to my brothers and sisters whom I missed so much: Devie, Syarif, Lola, Farhan, and Debby. Lastly, I would like to mention my lovely children, who are the *byproducts* of my Ph.D., making the journey more unique, memorable, and colorful: Fathimah and Fatih. At the time of writing, the latter is still inside the womb. I hope this dissertation inspires them to do better in their future endeavor than their father did. Until then, I will enjoy what's next in my post-Ph.D. journey!

Preface

The following two paragraphs are adapted from my statement of purpose when I applied for a Ph.D. at UT Austin in 2018.

> “*It all began two years ago when I worked as an integrated circuit physical design engineer at one of the fabless semiconductor companies. It was 11:37 p.m. that night when I suddenly awoke after having been asleep on my desk for one or two hours. The console on my screen had drawn my attention and made me gasp for a while, seeing that my place and route using an EDA tool had not been done. I was helping the wireless research team develop multi-bit latches and multi-bit FF for advanced process nodes, and I had to report my critical findings in a meeting with the U.S. team the following day. At that time, I wondered why I could not reduce the turnaround time to a greater extent even though I had reserved more cores on the grid server.*
>
> *Since my first contact with them, I have dreamed of creating a faster computer system. Almost every time I stare at a progress bar on my computer, I ask myself, ”How could I create a better and faster computer?”. Even though the terms “better” and “faster” are still general, the calling becomes stronger when I feel that the increase in computing performance has encountered stagnancy over the last decades as it already hit the three walls of limitations: power, complexity, and memory. This phenomenon has played a critical role in paving my life path to my field of interest: high-performance computer system architectures.*”

The above paragraphs show a few of my endeavors. Unfortunately, in my country, Indonesia, the industry is not there, let alone the research in this area. I need to go somewhere to fulfill my curiosity. Upon consulting with my undergraduate professor, Prof. Yudi Satria Gondokaryono, he suggested I apply for a Ph.D. at UT

Austin. Following his advice, I applied, and I was admitted to the Ph.D. program in Electrical and Computer Engineering at UT Austin. I was happy, but at the same time, I was afraid. I never did any research related to computer architecture during my undergraduate studies. I saw my friends were more ready for research in this field than I was. I needed to make extra efforts to catch up with research while trying to survive the graduate courses I took during the first four semesters.

I am so lucky to have Prof. Lizy K. John as my advisor. I learned so much about the foundation of performance evaluation of computer systems and workload characterizations. Since I am deeply interested in Graphics Processing Units (GPUs), Prof. John gave me research topics related to GPUs. GPUs have been my friend since high school, primarily for playing games and running Folding@Home. Accelerating applications using GPUs is one of my research interests. Thanks to Dell Technologies and the Texas Advanced Computing Center (TACC), I was so excited to have access to the latest and greatest GPUs, which I would not get anywhere in Indonesia. The project that becomes part of this dissertation is also about GPUs, a collaboration with ExxonMobil Technology and Engineering Company.

My dissertation discusses the novel algorithm of wave simulation using the discontinuous Galerkin (dG) discretization method and how to accelerate the simulation using modern hardware. Wave simulations are used in many applications, both in industry and academia. The dG method promises lower communication costs than Finite Difference Method (FDM) and Spectral Element Method (SEM), which are usually used in the industry. Combined with the Gauss-Lobatto-Legendre (GLL) integration scheme on straight-faced hexahedral elements, the number of BLAS operations is significantly reduced, consisting of only Level-1 BLAS operations. However, the CPU version of the code, provided by my colleagues at ExxonMobil Technology and Engineering Company, still consumes considerable simulation time, even though hundreds of CPU cores in HPC clusters are used. Since these simulations are often run for finding millions of wave solutions, accelerating the simulation and reducing the time-to-solution is critical.

With wave simulations' enormous parallelism, GPUs have become the most viable hardware for accelerating them. This is where the excitement begins! I started with a small-scale problem first that fits inside the memory of single GPU, 16 GB of NVIDIA Tesla V100. I rewrote the simulation kernels that perform volume, flux, and integration computation on CUDA as standalone codes, isolating them from the library dependencies. The basic GPU implementation, while significantly faster than two 24-core Intel Xeon Platinum 8160 CPUs, is still far away from the peak performance of the GPU. So, it is time to identify the bottleneck by profiling it using `nvprof`. Upon profiling, the wave simulation is memory-bound even with 900 GB/s of HBM2 memory on the NVIDIA Tesla V100 GPU. The next step is to perform architecture-aware optimizations of the wave simulation, described in detail in my dissertation. The optimization techniques used here can be generalized for other memory-bound workloads that run on GPUs. Although performance improvements after optimization are highly satisfactory, the wave simulation is still memory-bound.

Another issue with the GPU version is that it needs to be scalable across multiple GPUs to handle larger problem sizes. The CPU codes use `p4est` to partition and distribute the mesh across CPUs where Message Passing Interface (MPI) is used to perform synchronization of data (i.e., ghost exchange) between CPUs. Unfortunately, `p4est` does not support GPUs, and thus, I have either to build the mesh handling library from scratch or add GPU support to `p4est`. The latter makes more sense and saves significant development time. I implemented ghost exchange mechanisms on GPUs, which can utilize NVIDIA GPUDirect RDMA technology for inter-GPU communication through InfiniBand and CUDA-Aware MPI that supports asynchronous progression, achieving near-perfect weak scaling.

Even though optimization techniques have been applied, GPU implementation still suffers from intra- and inter-device communication bottlenecks. Therefore, I devised several strategies to reduce the communication overhead of the GPU implementation and improve its overall performance. The first part aims to reduce the intra-device communication overhead between the GPU's on-chip and off-chip

memory. In contrast, the second part aims to reduce inter-device communication in multi-GPU implementation by reducing the amount of data transferred during the ghost exchange. Again, the methods presented here are generic and can be used for other scientific applications.

Finally, in addition to GPU, I explored processing-in-memory (PIM), an emerging computing technology that allows computation to be performed where the data is stored. PIM promises to solve the memory wall with modern hardware architecture, especially for memory-bound workloads, since the data does not need to be brought from off-chip to on-chip memory to perform the computation, significantly reducing the data movement overhead. Thus, using PIM for dG-based wave simulation could improve the simulation performance and increase energy efficiency. In fact, to the best of my knowledge, this is the first effort of utilizing PIM for wave simulations. However, being the first adopter has its penalties; no compiler or software stack can be used to port the applications to utilize PIM. Therefore, I have to map the wave simulation manually, from optimizing the layout of the data on memory to maximizing parallelism to synchronizing data between memory blocks. It is tedious work and difficult to debug; however, the results outweigh the efforts, as described in detail in this dissertation. I hope that exploring PIM in this dissertation sets an example of the feasibility of PIM in accelerating high-performance scientific applications. Thus, more people are interested in researching this area, building ecosystems, and making PIM more accessible to the general public.

To wrap up, this dissertation is the product of my Ph.D. journey for six and a half years. After all of this time, I am happy to say that, with this dissertation, I have achieved most of what I described in my statement of purpose. Of course, I will not stop here; I am continuing my research in this area after getting my Ph.D., helping advance computing one step at a time. I hope the readers find it helpful and enjoy reading every part as much as I enjoyed my Ph.D. journey. As an Indonesian proverb describes, *no ivory is not cracked*, I understand this dissertation is imperfect. Thus, I would be more than happy to receive any feedback or questions from the readers.

Abstract

Algorithmic and Architectural Optimizations for Discontinuous Galerkin Finite Element Simulations

Bagus Hanindhito, PhD
The University of Texas at Austin, 2025

SUPERVISOR: Lizy Kurian John

Large-scale wave simulations are employed in various fields, including medical imaging, oil and gas exploration, earthquake hazard mitigation, and defense systems. Since these applications typically require repeatedly finding solutions of the wave equation on supercomputers, reducing both time-to-solution and energy-to-solution is crucial. This dissertation investigates the main bottlenecks in a class of wave simulations that utilize the dG finite element method with a GLL integration scheme on straight-faced hexahedral elements and proposes algorithmic and architectural optimizations to improve their performance. While this approach's computational cost is lower than the general mesh due to the reduced BLAS operations comprising only Level-1 BLAS, they are still too costly for many high-fidelity, industry-relevant applications. Furthermore, the existing implementation uses general-purpose CPUs, which may not be the most efficient hardware due to the considerable predictability in the execution flow, regularity in memory accesses, and, most importantly, the abundant parallelism that can be extracted.

The first contribution of this dissertation is accelerating the dG-based wave simulations using GPU. It is not a trivial task to port the CPU codes into GPU and achieve high speed-up, and thus, a set of hardware-informed algorithmic optimizations

are proposed and implemented. Compared to basic GPU implementation, fusing the kernel and using LUT for storing mesh structure yield up to 2.6× speed-up. Optimizing the utilization of shared memory and registers brings an additional 1.49× performance boost. In addition, multi-GPU support is implemented to support large-scale, industry-relevant problem sizes. Optimization strategies are proposed to minimize intra- and inter-node communication overhead, including a) reduction in data size for communication, resulting in reduced communication overhead by 70.27%, and b) the use of MPI with GPUDirect and asynchronous progression, further cutting overhead by 82.03%. With these optimizations, the GPU implementation of wave simulations achieves weak scaling across 128 GPUs.

The second contribution of this dissertation addresses the intra-device and inter-device communication overhead that limits the attained performance of dG-based wave simulations. It proposes a set of communication-reducing algorithms to reduce communication overhead between on-chip and off-chip memory (intra-device) and between the devices across many compute nodes (inter-device). The proposed algorithms can reduce intra-device data movement overhead resulting in performance improvements: 1) Node-tiling delivers an average performance improvement of 20%, with up to 37% better performance compared to the kernel without node-tiling and additional 15% added performance if shared memory is used optimally; 2) unified kernel enhances performance by an average of 22%, with additional 11% and 7% performance boost if shared memory is used optimally. The inter-device communication overhead is also reduced: 1) Face-Node-Only ghost exchange results in an average of 82% lower inter-device communication overhead compared to the existing CPU code; 2) Reduced-Precision ghost exchange results in 49.17% and 74.06% reduction of communication overhead, using single- and half-precision data types.

The third contribution of this dissertation is developing memristor-based PIM architecture to accelerate the dG-based wave simulations. PIM promises to alleviate the memory bottleneck in von Neumann architectures. The architectural enhancements to PIM tailored to support wave simulations are proposed, including the mem-

ory blocks interconnect, the look-up table for storing mesh structure, and complex computation offloading. In addition, optimization efforts are performed, including optimized data layout for reducing inter-memory block communications, batched execution to support larger problem sizes, expanded execution to increase row-parallel operations, and pipelining to improve the throughputs. These optimization efforts led to performance improvements and reduced data movement, achieving an average speed-up of 41.98× and energy savings of 12.66× over the GPU implementation.

Finally, it is important to note that efficient strategies for computing acoustic and elastic wave equations on GPUs and PIMs can also be applied to electromagnetic waves, given their structural similarities. The methods, strategies, and techniques discussed in this dissertation have broader applicability across various fields, highlighting the overall importance of this work.

Table of Contents

List of Tables 25
List of Figures 26
List of Algorithms 30
Chapter 1: Introduction 31
1.1 Problem Description 32
1.2 Motivation 33
1.3 Research Objectives 34
1.4 Thesis Statement 35
1.5 Contributions of this Dissertation 35
1.6 Dissertation Structure 36
Chapter 2: Background and Related Work 39
2.1 Seismic Survey 39
2.1.1 Generating and Recording Seismic Waves 40
2.1.2 Amount of Data 40
2.1.3 Modeling the Earth 41
2.2 Wave Equations 41
2.2.1 Acoustic Wave Equations 42
2.2.2 Elastic Wave Equations 43
2.2.3 Similarity with Other Hyperbolic PDEs 43
2.3 Mathematical Foundation for Computation 44
2.3.1 Discontinuous Galerkin Discretization 45
2.3.2 Gauss-Lobatto-Legendre Spatial Integration 47
2.3.3 Fourth-Order Low-Storage Runge-Kutta Temporal Integration . 49
2.4 Third-Party Software Libraries 50
2.4.1 Adaptive Mesh Refinement (AMR) Library 50
2.4.2 Message Passing Interface (MPI) Library 51
2.5 Analysis of Computation and Communication 55
2.5.1 Identifying Data Movement 56
2.5.2 Roofline Analysis 56
2.6 Graphics Processing Unit (GPU) and Its Applications 59
2.6.1 Road to Become General-purpose Massively-Parallel Accelerator 59
2.6.2 Hardware and Software Perspective 60

2.6.3 Memory Hierarchies . . . 66
2.6.4 The Inclusion of Matrix Accelerator . . . 71
2.6.5 Mixed-Precision Computation . . . 73
2.6.6 Previous Work on GPU-Accelerated Scientific Applications . . 73
2.6.7 Previous Work on GPU-Accelerated dG-based Discretization . 76
2.7 Processing-in-Memory (PIM) and Its Applications . . . 77
2.7.1 Analog PIM . . . 78
2.7.2 Digital PIM . . . 79
2.7.3 Applications Leveraging PIM . . . 81
Chapter 3: Methodology . . . 82
3.1 Wave Simulation Code Base . . . 82
3.1.1 CPU Codes Examination . . . 83
3.1.2 Porting and Optimizing Codes . . . 83
3.1.3 Results Verification . . . 85
3.2 Computing Platform and Infrastructure . . . 87
3.2.1 TACC Maverick2 Cluster . . . 87
3.2.2 TACC Longhorn Cluster . . . 89
3.2.3 TACC Lonestar6 Cluster . . . 93
3.2.4 Other Computing Resources . . . 93
3.3 Memristor-based Processing-in-Memory Models . . . 95
3.3.1 PIM Simulation Software . . . 95
3.3.2 Non-Volatile Memory Circuit Models . . . 96
3.3.3 Energy Models . . . 97
3.3.4 PIM Platform Configurations . . . 98
3.3.5 Process Node Scaling . . . 100
3.4 Measurement Tools and Computing Libraries . . . 100
3.4.1 Profiling GPU Kernels . . . 100
3.4.2 Measuring Power Consumption . . . 102
3.4.3 Measuring Performance . . . 103
3.4.4 Computing Libraries . . . 103

Chapter 4: Large-scale dG-based Wave Simulations on CPUs 104
4.1 Mesh Generation and Partitioning 105
4.1.1 Refinement of Meshes 106
4.1.2 Elements of Mesh 108
4.1.3 Element Numbering Schemes 110
4.1.4 Source of Parallelism 111
4.1.5 Partitioning and Distributing Mesh 111
4.2 Data Structure 112
4.2.1 Mesh Data 113
4.2.2 Mesh Structure 113
4.2.3 Element Database 117
4.3 Simulation Flow and Data Flow 118
4.3.1 Simulation Flow 119
4.3.2 Data Flow 120
4.4 Simulation Kernels 121
4.4.1 Mass-Inverse Kernel 121
4.4.2 Volume Kernel 122
4.4.3 Flux Kernel 123
4.4.4 Integration Kernel 124
4.5 Multi-CPU Implementation 125
4.5.1 Data Structure 125
4.5.2 Ghost Exchange 129
4.6 Result Verification and Performance Measurement 131
4.6.1 Numerical Accuracy 131
4.6.2 Solution Ranges 131
4.6.3 Runtime Performance 132
Chapter 5: Accelerating dG-based Wave Simulations using GPUs 133
5.1 Data Structure Modification for GPU 134
5.1.1 Mesh Data 136
5.1.2 Mesh Structure 136
5.2 Execution Flow Modification for GPU 142
5.2.1 Execution Flow for Interior of Elements 142
5.2.2 Execution Flow for Face of Elements 144
5.3 Basic GPU Simulation Kernels Implementation 146
5.3.1 Mass-Inverse Kernel 147

5.3.2 Volume Kernel 147
5.3.3 Flux Kernel 149
5.3.4 Integration Kernel 153
5.4 GPU Simulation Kernels Optimizations 153
5.4.1 Kernel Fusion 153
5.4.2 Shared Memory Utilization 156
5.4.3 Register Allocation Improvement 158
5.5 Performance Analysis 159
5.5.1 Simulation Runtime 159
5.5.2 Speed-up Over CPUs 162
5.5.3 Roofline Analysis 164
5.5.4 Effects of Element Order on Performance 165
5.5.5 Early Performance Exploration on Newer GPU Architecture . . 168
5.6 Multi-GPU Implementation and Optimization 169
5.6.1 Ghost Layer and Ghost Buffer Implementation 170
5.6.2 Ghost Exchange Optimization 172
5.6.3 CUDA-aware MPI with Asynchronous Progression Support . . 173
5.6.4 Scalability Evaluation 177
Chapter 6: Reducing Communication for dG-based Wave Simulations 179
6.1 Achieving dG Element Locality with Node Tiling 181
6.1.1 Ideal Execution for Harnessing dG Locality 181
6.1.2 Locality of Element and Hardware Limit 184
6.1.3 Modifying Execution Strategy using Node Tiling 188
6.2 Preserving GPU Execution States using Unified Kernel 191
6.2.1 Modifying The Simulation Flow 192
6.2.2 Data Hazard Problem 193
6.2.3 Naïve Double Buffering 197
6.2.4 Double-Buffering without Additional Buffer 199
6.2.5 Unified Kernel Implementation 202
6.3 Face-Nodes-Only Ghost Exchange 206
6.3.1 Modifying `ElementDataBase` Structure 207
6.3.2 Implementation Challenge 208
6.3.3 Array Index Mapping Strategy 208
6.3.4 Modifying Kernels to Support New Ghost Elements 212
6.4 Reduced-Precision Ghost Exchange 212

6.4.1 Multi-Precision Strategy 213
6.4.2 Code Implementation 216
6.5 Early Exploration of Partial Ghost Exchange 218
6.5.1 Approximation of Stress Values by Computing Displacements . 220
6.5.2 Approximation of Stress Values by Integrating Constitutive and Kinematics Equation 220
6.6 Intra-Device Communication Performance Evaluation 221
6.6.1 GPU Code Flavors 221
6.6.2 Simulation Runtime 223
6.6.3 Key Takeaways 227
6.7 Inter-Device Communication Performance Evaluation 227
6.7.1 Evaluating Face-Node-Only Ghost Exchange 228
6.7.2 Evaluating Reduced-Precision Ghost Exchange 233
6.7.3 Key Takeaways 237
Chapter 7: PIM Architecture for Accelerating dG-based Wave Simulations . 238
7.1 Architecture for Wave Simulation 240
7.1.1 Hierarchical Organization 240
7.1.2 Central Controller 242
7.1.3 Memory Block Interconnection 242
7.1.4 Look-up Table Implementation on PIM 247
7.1.5 Interaction with Host CPU and Off-chip Memory 249
7.1.6 Choice of The Data Types 249
7.1.7 Mapping Wave Simulation to PIM 250
7.2 Basic Implementation of Wave Simulation 251
7.2.1 Data Layout in Memory 251
7.2.2 Execution Timeline 253
7.3 Optimizations for Wave Simulation 256
7.3.1 Handling Large Problem Sizes with Batching 256
7.3.2 Increasing Parallelism with Expansion 260
7.3.3 Improving Throughput with Pipelining 263
7.4 Performance Analysis 265
7.4.1 Configurations for Evaluation 265
7.4.2 Performance Comparison Against GPU 269
7.4.3 Energy Consumption Comparison Against GPU 270
7.4.4 Pipelining Analysis and Evaluation 272
7.4.5 Memory Block Interconnect Evaluation 273

Chapter 8: Conclusion . . . 275
8.1 Summary . . . 275
8.2 Key Takeaways . . . 279
8.3 Future Work . . . 280
Appendix A: Mathematical Review . . . 285
A.1 The Acoustic Wave Equation . . . 285
A.2 The Elastic Wave Equation . . . 286
A.3 Discontinuous Galerkin (dG) Discretization . . . 287
A.3.1 Overview of dG . . . 287
A.3.2 Discretizing Problem Domain . . . 288
A.3.3 Higher Order Element . . . 289
A.3.4 Discontinuity of Solutions . . . 290
A.3.5 Deriving Continuous Weak Form . . . 290
A.3.6 Discretization in Space . . . 293
A.3.7 Discretization in Time . . . 297
A.4 Example of Discontinuous Galerkin (DG) in Acoustic Wave . . . 298
A.4.1 Deriving The Continuous Weak Form . . . 298
A.4.2 Discretization in Space . . . 300
A.4.3 Discretization in Time . . . 302
Appendix B: Detailed Explanation of Wave Simulation Codes . . . 303
B.1 Element Data Structure . . . 303
B.2 Simulation Output . . . 306
B.2.1 Runtime Output . . . 306
B.2.2 Diagnostic and Statistic Output . . . 307
B.2.3 Visualization Output . . . 309
B.3 Main Program Execution Flow . . . 310
B.4 Kernel Execution Flow . . . 315
B.4.1 Mass Inverse Kernel . . . 315
B.4.2 Volume Kernel . . . 317
B.4.3 Flux Kernel . . . 320
B.4.4 Integration Kernel . . . 328
B.5 Compile-time Configuration . . . 328
B.5.1 Compilation Dependencies . . . 330
B.5.2 Compilation Properties . . . 330
B.5.3 Simulation Properties . . . 331

B.5.4 GPU-Acceleration Support 332
B.5.5 Optimized Communication Support 333
B.6 Runtime Configuration 333
Glossary 337
References 374
Vita 410

List of Tables

2.1 The Terminology Equivalency between NVIDIA, AMD, and Intel GPUs 63
2.2 Digital PIM Multiplication Latency 80

3.1 TACC Computing Cluster Configurations 88
3.2 Energy and Time Parameter for PIM Basic Operations 96
3.3 The Read Latency Breakdown of PIM from NVSim 96
3.4 The Energy Model for PIM Chip with 2 GB Capacity 97
3.5 PIM Platform Configurations 99

4.1 Size of Element Data Structure (`ElementDataBase`) 118

5.1 GPU Kernel Configuration Flavors for Performance Evaluation 159
5.2 Standard and Ghost Element Size 173
5.3 Comparison of MPI Library Implementations 173

6.1 Standard, GhostV1, and GhostV2 Element Size 207
6.2 Standard, GhostV1, and GhostV2 Element Size with Reduced-Precision 217
6.3 GPU Kernel Configuration Flavors for Performance Evaluation 222

7.1 Comparison of Accuracy for Double and Single-Precision Arithmetic in PIM 250
7.2 Workloads Characteristic for Evaluating PIM Performance 266
7.3 PIM Configuration Flavors for Performance Evaluation 268

B.1 The Contents of `ElementDataBase` 305

List of Figures

1.1 Seismic Survey and Problem Domain Discretization 31

2.1 Unknown Variables in Acoustic and Elastic Wave Simulation 42
2.2 Communication pattern of DG, FDM, and SEM 46
2.3 Uniform Mesh and Non-Conforming Mesh 46
2.4 Node Clustering on Element with GLL Quadrature 49
2.5 Example of Roofline Chart 58
2.6 The Organization of GPU from Software and Hardware Perspective . 61
2.7 Simplified Diagram of Streaming Multiprocessor in NVIDIA GPU . . 64
2.8 The Logical View of Memory Hierarchy in GPUs 68
2.9 The Physical View of Memory Hierarchy in GPUs 72
2.10 Bitwise NOR Operation using Memristor 80

3.1 Comparing Result of CPU and GPU Codes for Verification 86
3.2 Intra-Node Interconnect Topology of TACC Longhorn Compute Node 91
3.3 Inter-Node Interconnect Topology of TACC Longhorn Cluster (Approximated) 92
3.4 Intra-Node Interconnect Topology of DGX-A100 Platform 94

4.1 Problem Domain Discretization 105
4.2 Refinement Level of Discretization of Problem Domain 107
4.3 The Nodes Configuration Inside Element 109
4.4 Numbering Scheme of Elements 110
4.5 Mesh Partition and Distribution by `p4est` 112
4.6 Data Structures of `p4est` for Managing Mesh Data and Mesh Structure 114
4.7 High-level Execution Diagram of Volume Computation by `p4est` . . . 115
4.8 High-level Execution Diagram of Flux Computation by `p4est` 116
4.9 Simplified High-Level Flow of Wave Simulation Application 119
4.10 Simplified High-Level Data Flow of Element 121
4.11 Ghost Layer Illustration 126
4.12 Ghost Buffer Illustration 128

5.1 Look-up Table (LUT) for Finding Elements in GPU 141

5.2 High-level Execution Diagram of Volume Computation in GPU . . . 143
5.3 High-level Execution Diagram of Flux Computation in GPU 145
5.4 Modified Simulation Flow for Implementing Fused Kernels in GPU . 155
5.5 Look-up Table (LUT) for Finding Nodes on Element's Faces in GPU 156
5.6 Simulation Time on GPU 160
5.7 Speed-up of Intel Xeon CPUs over Baseline IBM POWER9 CPUs . . 163
5.8 Speed-up of single NVIDIA Tesla V100 GPU over Baseline IBM POWER9 CPUs 163
5.9 Roofline Analysis for Simulation Kernels on GPU 166
5.10 The Effect of Element-Order to Performance Simulation 167
5.11 Roofline Analysis for Wave Simulation with Different Element-Order 167
5.12 Simulation Performance on NVIDIA Tesla V100 and NVIDIA A100 GPUs 169
5.13 Implementation of Ghost Layer and Ghost Buffer in GPU for Ghost Exchange 171
5.14 Multi-GPU Simulation Performance with Ghost Exchange Optimization 174
5.15 Comparison of Multi-GPU Simulation Performance on Various MPI Libraries 176
5.16 Multi-Node Multi-GPU Simulation Performance Scalability 178

6.1 Ideal Thread-Block Execution for Element 183
6.2 SM-Occupancy-Aware Thread-Block Execution for Element 186
6.3 Node-Tiling Thread-Block Execution for Element 189
6.4 Comparison of Execution Flow Inside Element for Ideal, SM-Occupancy-Aware, and Node-Tiling 190
6.5 Modified Simulation Flow for Implementing Unified Kernels in GPU . 193
6.6 The Inter-Element Data Hazard When Fusing Integration Kernel with Volume and Flux 196
6.7 Naïve Double Buffering Implementation to Avoid Inter-Element Data Hazards 198
6.8 Optimized Double Buffering with Buffer-Swapping 200
6.9 The Execution Flow of Unified Kernel 205
6.10 Exterior Facing Nodes of Elements 206
6.11 The Node Indexing Challenge for `ElementDataBaseGhostV2` 209
6.12 Mapping Example 211
6.13 Various Strategy for Reduced-Precision Ghost Exchange 215
6.14 Code Clutter with Explicit Typecasting 218

6.15 Ghost Multi-Precision C++ Class with Overloaded Operators 219
6.16 Simulation Time on GPU with Communication-Reducing Strategy . . 224
6.17 Multi-GPU Simulation Performance with Face-Node-Only Ghost Exchange 229
6.18 Additional Compute Overhead with Face-Node-Only Ghost Exchange 231
6.19 Early Multi-GPU Wave Simulation Performance Exploration with DGX-A100 Platform 232
6.20 Multi-GPU Simulation Performance with Reduced-Precision Ghost Exchange 234
6.21 Comparing Performance and Numerical Accuracy of Reduced-Precision Ghost Exchange 236

7.1 PIM Architecture and Its Hierarchical Organization 241
7.2 H-Tree Interconnect Topology for PIM 244
7.3 Bus Interconnect Topology for PIM 246
7.4 LUT Instruction Format for ISA-based PIM 248
7.5 Memory Data Layout for Element of Mesh in PIM 252
7.6 Volume and Flux Kernels Execution Timeline in PIM 254
7.7 Batching Mechanism for Volume and Integration Kernels 257
7.8 Batching Mechanism for Flux Kernels 259
7.9 Expansion Mechanism for Volume Kernel 261
7.10 Expansion Mechanism for Flux Kernel 263
7.11 Pipelined Execution Mechanism for Wave Simulation Kernels in PIM 264
7.12 Performance Comparison of PIM and GPU 270
7.13 Energy Consumption Comparison of PIM and GPU 271
7.14 The Execution Time Breakdown of Pipelining in PIM 272
7.15 Performance Comparison of H-Tree and Bus Interconnect on PIM . . 273

A.1 Example of discretization in 1D space 288
A.2 Example of discretization in 2D space 289
A.3 Element boundary of dG discretization 292
A.4 Discretization in space and polynomial approximation 294

B.1 Runtime Output of Acoustic Wave Simulation 306
B.2 Runtime Output of Acoustic Wave Simulation 307
B.3 Diagnostic Output of Acoustic Wave Simulation 308
B.4 Runtime Output of Elastic Wave Simulation 309

B.5 Visualization of Wave Simulation Output 311
B.6 Detailed High-Level Flow of Wave Simulation Application 312
B.7 Detailed High-Level Flow of Wave Simulation Engine Initialization . . 314
B.8 Detailed High-Level Flow of Mass Inverse Kernel 316
B.9 Detailed High-Level Flow of Volume Kernel (Generic) 318
B.10 Detailed High-Level Flow of Flux Kernel (Generic) 321
B.11 Detailed High-Level Flow of Flux Vector Functions (Generic) 323
B.12 Detailed High-Level Flow of Integration Kernel 329

List of Algorithms

1 Volume kernel 122
2 Flux kernel 124
3 Iterative Method to Find Neighboring Elements 139
4 Basic Flux Kernel Execution on GPU 150
5 Iterative Method to Find Face Node Index 152
6 Calculate `ElementDataBaseGhostV2` Offset Index from `ElementDataBase` Node Index 210
7 Execute Single LUT Instruction 248

Chapter 1: Introduction

Wave simulations have become an important topic in science and engineering, with a wide range of applications. This dissertation investigates the performance of discontinuous Galerkin (DG) discretizations in large-scale wave simulation. Specifically, it uses the Gauss-Lobatto-Legendre (GLL) integration scheme on straight-faced hexahedral elements to reduce the computational cost compared to general meshes. However, they are still prohibitively expensive for many high-fidelity, industry-relevant applications. With its unique challenges, modern GPUs and emerging architectures, such as Processing-in-Memory, can potentially make such high-fidelity simulations feasible.

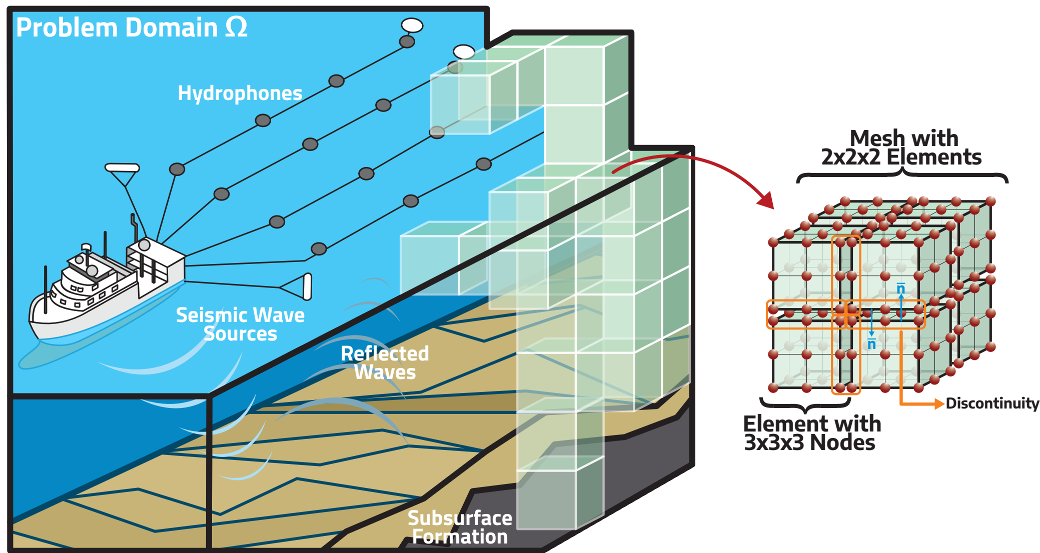


Figure 1.1: A marine seismic survey over a large area is conducted to gather data for full-waveform inversion by creating seismic waves. Hydrophones capture these waves as they reflect. The survey area must be discretized into a mesh containing many smaller elements, each with multiple nodes, for computer simulations.

This chapter provides a brief introduction to the research subject of this dissertation. It describes the context of the problem (Section 1.1) and the motivation

for conducting research in this area (Section 1.2). From these, the objectives of the research are formulated (Section 1.3) along with the thesis statement (Section 1.4). Finally, it provides a list of contributions of this dissertation (Section 1.5) and the structural organization of this dissertation document (Section 1.6).

1.1 Problem Description

Full-wavefield inversion, as described by Lacasse et al. (2018); Fathi et al. (2016), is a cutting-edge technique used to determine the material properties of the Earth's subsurface. These estimated properties provide various opportunities, including locating regions with hydrocarbons (useful for oil and gas exploration), identifying safe areas for carbon sequestration, and selecting suitable sites for hydrogen storage, a potential low-carbon energy source. Full-waveform inversion was not applied to realistic scenarios in the past due to its high computational demands, but the advent of petascale computing has changed that. The enhanced computational power provided by modern and emerging architectures can significantly improve the resolution and accuracy of subsurface imaging.

In the full-wavefield inversion, seismic waves are probing tools for examining the subsurface, as discussed by Sengbush (1983). These waves are artificially generated, typically at the surface of the unknown medium, before they propagate through it. Since the subsurface of the Earth comprises heterogeneous materials, the waves change their amplitude and direction. Some reflected waves return to the surface, where receivers detect them. An example of such a survey is illustrated on the left side of Figure 1.1, where the seismic waves are generated from a ship, and the reflected waves are recorded using hydrophones mounted on the back of the ship. The goal of this survey is to estimate the subsurface material properties using known input waves, measured output waves at receiver points, and a model describing the wave-motion physics in the subsurface. Using the computer, wave simulations are run by discretizing the problem domain into a mesh comprising a large number of elements,

as shown on the right side of Figure 1.1. This iterative process requires repeated wave-motion simulations, as shown by Fathi et al. (2015a), involves a vast amount of data (Section 2.1.2), and is computationally expensive.

With their importance and versatility in many applications, including seismic surveys (Section 2.1), accelerating large-scale wave simulations to reduce the time to solutions is essential. One way to accelerate the computation is by exploiting wave simulations' physical or geometry characteristics. For example, extracting as much parallelism that is abundant in wave simulations (Section 2.3.1) and mapping the computation to massively-parallel architecture, such as GPUs (Section 2.6) and PIMs (Section 2.7), can yield significant performance improvements.

1.2 Motivation

Large-scale wave simulations present new challenges in accelerating their computation due to their computational costs and the vast amount of data involved. It is used for many industry-relevant applications, as shown in the following work: seismic hazard mitigation by Poursartip et al. (2017), medical imaging by Guasch et al. (2020); Lucano et al. (2016), deriving the composition and features of the Earth's subsurface by Lacasse et al. (2018); Kallivokas et al. (2013); Mondol (2015), oceanography by Duda et al. (2019), and military by Hong et al. (2004).

A novel wave simulation algorithm utilizing the discontinuous Galerkin (dG) discretization method (Section 2.3.1) has been developed. The dG method offers lower communication overhead compared to the commonly used Finite Difference Method (FDM) and Spectral Element Method (SEM) in the industry. When paired with the Gauss-Lobatto-Legendre (GLL) integration scheme (Section 2.3.2) on straight-faced hexahedral elements, the number of BLAS operations is significantly minimized, consisting solely of Level-1 BLAS operations. While this algorithm is promising, its performance pattern on actual hardware remains to be seen.

The existing implementation of this algorithm uses general-purpose CPUs to

perform the wave simulation. Despite utilizing hundreds of CPU cores in HPC clusters, the CPU version of the code still requires substantial simulation time. Given that these simulations often involve solving millions of wave equations, speeding up the process and minimizing the time-to-solution and energy-to-solution is essential. CPUs might not be the most efficient hardware for running wave simulations due to their predictable execution flow, regular memory access patterns, and, most critically, the high level of parallelism that could be better exploited with other hardware.

Massively parallel architecture, such as GPUs and PIM, are most suitable for accelerating wave simulations based on their characteristics. Having a deep understanding of the existing CPU code, the GPU and PIM implementations can be developed. However, it is not a trivial task; it requires careful algorithm design, and often algorithmic and architectural optimizations to achieve the highest performance possible from this hardware while maintaining numerical accuracy. Furthermore, when dealing with vast amounts of data across large computing clusters, data movement often becomes the key bottleneck that limits the overall simulation performance, and thus, strategies to reduce the communication overhead must be developed.

1.3 Research Objectives

The research begins with studying the existing CPU codes of the dG-based wave simulation to identify its execution pattern, data flows, and the critical functions used to run the simulation. Based on these findings, the GPU implementation of the dG-based wave simulations is developed. However, the first version of the codes may not achieve satisfactory performance from the hardware, and thus, workload characterization must be performed to identify the key bottleneck. Then, algorithmic optimizations are developed to improve the performance of the wave simulations on GPUs. Since data movement often becomes the critical bottleneck of many workloads, communication-reducing algorithms are applied to reduce the communication overhead. Finally, the efficacy of PIM in further reducing the data movement over-

head in wave simulation is investigated. Architectural enhancements for PIM and optimization strategies are proposed to run large-scale dG-based wave simulations using this emerging computing technology.

1.4 Thesis Statement

Algorithmic optimizations and hardware-architectural enhancements can improve the performance and efficiency of dG-based large-scale wave simulations based on their performance patterns. Algorithmic optimizations, such as kernel fusion, LUT-based mesh structure, and improved on-chip memory usage, and architectural enhancements, such as batching, expansion, and pipelining, and communication reduction strategy, such as reduced precision data exchange, enable wave simulations to achieve performance and energy improvements on massively parallel architecture, including Graphics Processing Units (GPUs) and memristor-based Processing-in-Memory.

1.5 Contributions of this Dissertation

This dissertation proposes techniques to accelerate dG-based large-scale wave simulation using GPUs and memristor-based PIM. It discusses the strategies to perform algorithmic and architectural optimizations based on their execution patterns and the critical bottleneck. Specifically, the primary contributions of this dissertation can be broken down into three, as discussed below.

1. **Accelerating dG-based Wave Simulations using GPUs:** This contribution, presented in Chapter 5, aims to accelerate the wave simulations using GPU by performing necessary modifications to the codes and libraries, making them more GPU-friendly. Based on their performance patterns, this contribution also proposes algorithmic optimization methods to improve the performance of the wave simulations when running on the GPU. Furthermore, multi-GPU support

is developed and optimized to support larger problem sizes used in the industry, allowing the wave simulations to achieve weak scaling over 128 GPUs.

2. **Reducing Communication for dG-based Wave Simulations:** This contribution, presented in Chapter 6, addresses the crucial issues limiting wave simulations' overall performance on GPUs: intra-device and inter-device communications. This contribution proposes a set of communication-reducing algorithms to minimize the overhead of intra-device and inter-device data movement by improving the locality, reducing data movement, and lowering the data volume being exchanged.

3. **PIM Architecture for Accelerating dG-based Wave Simulations:** This contribution, presented in Chapter 7, enables running dG-based wave simulations on memristor-based Processing-in-Memory, an emerging computing technology promising to alleviate memory bottleneck on von Neumann architectures. This contribution proposes architectural enhancements to the PIM tailored for running wave simulations. Along with optimized data placement and execution strategy, the PIM implementation achieves significant speed-up and energy savings compared to the GPU implementation.

1.6 Dissertation Structure

Following the front matters, this dissertation comprises eight main chapters, including this chapter, and two appendices. The main chapters of this dissertation are organized as follows. This dissertation begins with Chapter 1, introducing the reader to the research problem as the main topic, the research motivation and objectives, the thesis statement, and the contributions of this dissertation. Next, Chapter 2 lays the groundwork for the research presented in this dissertation. It familiarizes readers with the key theories and concepts necessary to grasp the work discussed in later chapters. Additionally, it offers a brief overview of prior research relevant to the dissertation's

topic. While it summarizes the existing knowledge, readers are encouraged to consult the cited references for more in-depth information. Interested readers are suggested to refer to Appendix A for detailed discontinuous Galerkin (dG) formulation for seismic wave simulations. Following is Chapter 3, which thoroughly explains the research methods used in this dissertation. It outlines the methodologies, hardware platforms, software libraries, and tools employed throughout the study.

As an entrance to the discussion of the dissertation's contributions, Chapter 4 briefly explains the existing dG-based wave simulation CPU code, which becomes the baseline code base of the research in this dissertation. This chapter is accompanied by Appendix B, which thoroughly explains the CPU code base. Next, Chapter 5 describes the first contribution of this dissertation: accelerating the dG-based wave simulations using Graphics Processing Units (GPUs). It provides a detailed explanation of the required modifications to provided code bases, mapping the problems into the GPU, characterizing, evaluating, and optimizing the GPU codes, and improving the scalability of the GPU codes.

Continuing into the second contribution, Chapter 6 describes algorithms to reduce the intra-device and inter-device communications in the dG-based wave simulations on GPUs. It provides algorithm innovations and ideas to minimize intra-device data movement, which the previous chapter has shown as the critical bottleneck in dG-based wave simulations. It also describes the techniques to reduce inter-device communication, often limiting large-scale wave simulation performance on large computing clusters.

Last but not least, Chapter 7 describes the third contribution of this dissertation: accelerating the dG-based wave simulations using Processing-in-Memory (PIM), an emerging computing technology. It provides the detailed PIM architecture implementation, mapping the problems into the PIM, improving the PIM performance, energy efficiency, and scalability through several optimization techniques, and evaluating PIM against the GPU implementations.

Finally, Chapter 8 concludes this dissertation; it summarizes the work done as contributions to this dissertation and discusses some potential ideas for future work in this area.

Chapter 2: Background and Related Work

This chapter establishes the foundation for the research in this dissertation. It gives the reader some familiarity with the underlying theorems and knowledge, helping them understand the work described in the subsequent chapters. It also briefly discusses previous works related to the topic of this dissertation. While it provides a comprehensive overview of the existing knowledge related to this dissertation, interested readers are always suggested to read the given references for more detailed explanations.

First, the seismic survey is briefly discussed in Section 2.1, which gives insight into the efforts and challenges of modeling the Earth's subsurface through seismic waves. Next, the underlying wave equations, including acoustic and elastic wave equations, are briefly discussed in Section 2.2. In addition, the mathematical foundation for performing the wave simulation is briefly covered in Section 2.3. This includes the discontinuous Galerkin discretization method, the Gauss-Lobatto-Legendre spatial integration, and the Low-Storage Runge-Kutta temporal integration. Appendix A provides an in-depth explanation from a mathematical perspective. Two important third-party software libraries for developing the wave simulations are discussed in Section 2.4. These two libraries are the adaptive mesh refinement library provided by `p4est` and the message-passing interface library. Next, the analysis of computation and communication using the roofline method is discussed in Section 2.5. Finally, an overview of the hardware architectures used in this dissertation is provided: Graphics Processing Unit in Section 2.6 and Processing-in-Memory in Section 2.7.

2.1 Seismic Survey

Seismic surveys have been becoming the primary method for approximating the subsurface properties of the Earth since they have lower exploration costs by

eliminating the need for digging, drilling, or tunneling as described by Sengbush (1983); Mondol (2010). The survey uses the generated seismic waves that travel into the Earth. Subsurface formations reflect the waves, which are then recorded for further analysis. This technique is called reflection seismology, which, carefully speaking, is an older method for oil and gas exploration as described by Bates et al. (2016). It was wildly popular before the development of petascale supercomputers. Since the development of more powerful supercomputers, a more rigorous technique called full-waveform inversion has become more attractive.

2.1.1 Generating and Recording Seismic Waves

Artificial sources, such as air guns in marine environments or dynamite in land environments, are used to generate seismic waves for the survey. The reflected waves are acquired using hydrophones or geophones, as described by Mondol (2010). The former is used in marine environments and detects seismic energy by sensing the pressure changes in water. In contrast, the latter is used in the onshore and offshore seabed to detect the ground velocity generated by the seismic waves. As mentioned by Haldar (2018), the subsurface of the Earth comprises different materials, which have different wave-reflection and wave-refraction properties. Utilizing this fact, the recorded reflection is analyzed to approximate the composition of Earth's subsurface. An accurate approximation of the Earth is highly valuable for identifying the depository of natural resources, such as oil and gas reservoirs.

2.1.2 Amount of Data

While seismic surveys are helpful tools for understanding the composition of Earth's subsurface, they generate vast amounts of data. A typical seismic survey in oil and gas exploration covers hundreds of kilometers square and reaches 10 kilometers deep. An example given by Lacasse et al. (2018) considers a seismic survey conducted using a ship covering an area of 40 km by 40 km. Each shot of seismic waves is

generated by triggering the seismic source every 50 m while the ship moves, and hence, 40,000 shots are needed to cover this area. The reflected waves are recorded by the hydrophones, which are placed every 12.5 m in eight 8 km cables. This means every seismic shot will generate 5,120 seismograms. Using a sampling period of 2 ms and 24-bit data formats, this survey easily generates 3.1 TB of data.

2.1.3 Modeling the Earth

There are two problems related to finding the best Earth model utilizing data obtained from seismic surveys. The first problem is called the forward problem, which is the focus of this dissertation. The forward problem aims to generate synthetic seismograms based on the defined 3D Earth model as explained by Lacasse et al. (2018); Fathi et al. (2015b). The second problem is called the inverse problem, whose objective is to find the optimum model of the Earth that best describes the data obtained from the seismic surveys, as explained by Lacasse et al. (2018); Fathi et al. (2015b); Kallivokas et al. (2013); Fathi et al. (2016). In other words, this problem can be described as "optimizing the optimizer" using constrained partial differential equation (PDE).

2.2 Wave Equations

This dissertation considers acoustic and elastic wave equations. Finding the numerical solution of acoustic and elastic wave equations is important in many applications. They are widely used in oil and gas exploration (e.g., as shown by Lacasse et al. (2018)), earthquake hazard mitigation (e.g., as indicated by Poursartip et al. (2017)), site characterization of critical components of civil infrastructure (e.g., as described by Kallivokas et al. (2013); Fathi et al. (2016)), oceanography (e.g., as explained by Duda et al. (2019)), medical imaging (e.g., as demonstrated by Guasch et al. (2020)), and defense systems (e.g., as described by Hong et al. (2004)).

In summary, the work done in this dissertation considers three-dimensional

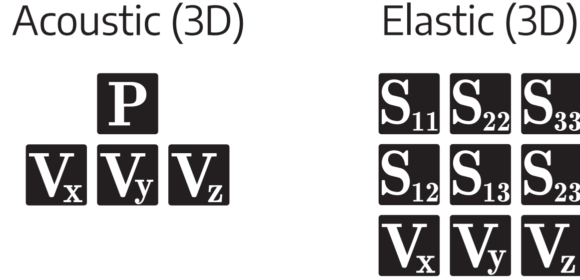


Figure 2.1: The unknown variables in acoustic and elastic wave simulation for 3D space. In 3D space acoustic wave simulation, there are four unknown variables: pressure p and particle velocity vector ($\mathbf{v}_x$, $\mathbf{v}_y$, and $\mathbf{v}_z$). In 3D space elastic wave simulation, there are nine unknown variables: second-order stress tensor ($\mathcal{S}_{11}$, $\mathcal{S}_{22}$, $\mathcal{S}_{33}$, $\mathcal{S}_{12}$, $\mathcal{S}_{13}$, and $\mathcal{S}_{23}$) and particle velocity vector ($\mathbf{v}_x$, $\mathbf{v}_y$, and $\mathbf{v}_z$).

space with four unknown values for acoustic wave equations (p, $\mathbf{v}_x$, $\mathbf{v}_y$, and $\mathbf{v}_z$) and nine unknown values for elastic wave equations (S_{11}, S_{22}, S_{33}, S_{12}, S_{13}, S_{23}, $\mathbf{v}_x$, $\mathbf{v}_y$, and $\mathbf{v}_z$). These variables are shown in Figure 2.1 and are evaluated at each discrete point in 3D space for every time step.

2.2.1 Acoustic Wave Equations

The equations of acoustic waves approximate the propagation of compressional waves in water, body tissue, and Earth. Acoustic wave problem is a subset of elastic wave problems, and they are written as partial differential equations (PDEs) as shown in Equation (2.1).

$$\frac{\partial p}{\partial t} + \kappa \nabla \cdot \mathbf{v} = 0, \tag{2.1a}$$

$$\frac{\partial \mathbf{v}}{\partial t} + \frac{1}{\rho} \nabla p = 0, \tag{2.1b}$$

Two unknown variables are pressure $p = p(x, y, z, t)$ and particle velocity $\mathbf{v} = \mathbf{v}(x, y, z, t)$. The $\mathbf{v}$ is a vector containing velocity values in the x, y, and z directions in the case of three-dimensional space, denoted as $\mathbf{v}_x$, $\mathbf{v}_y$, and $\mathbf{v}_z$, respectively. Bulk modulus and density define the material's properties and are denoted by κ and

ρ, respectively. The $\nabla\cdot$ and ∇ are divergence and gradient operators, respectively. Further discussion on acoustic wave equations is given in Appendix A.1.

2.2.2 Elastic Wave Equations

The elastic wave equations describe the propagation of compressional and shear waves in elastic solids. Equation (2.2) shows the PDEs used to describe elastic wave problems.

$$\frac{\partial \mathcal{S}}{\partial t} = \mu(\nabla \mathbf{v} + \nabla \mathbf{v}^T) + \lambda \nabla \cdot \mathbf{v}\ \mathcal{I}, \tag{2.2a}$$

$$\frac{\partial \mathbf{v}}{\partial t} = \frac{1}{\rho}\nabla \cdot \mathcal{S}, \tag{2.2b}$$

Two unknown variables are stress, described using stress tensor $\mathcal{S}$, and particle velocity, described using velocity vector $\mathbf{v}$. The $\mathcal{I}$ is the identity tensor. Material properties are described using Lamé parameters λ and μ, and material density ρ. The elastic wave equations are discussed further in Appendix A.2.

2.2.3 Similarity with Other Hyperbolic PDEs

As shown by Quarteroni and Valli (1994), acoustic and elastic wave equations belong to the broader hyperbolic partial differential equations (PDEs) class. When developing computational algorithms to obtain numerical solutions, Hyperbolic PDEs share many similarities: they have common operations[1], most of which involve Level-1 BLAS (i.e., vector-vector operations), and they have local communication patterns.

[1] These operations include:

a The problem domain is discretized into many smaller elements/cells, each with several nodes (Section 4.1). The objective of the computational algorithm is to find the solution field on these nodes.

b The computation of derivatives for certain fields must be done within each element. For tensor-product elements (e.g., hexahedral/cube used in this dissertation), the derivative computation involves a dot-product operation between the derivative vector and the subset of the element's nodes.

In this class, there are also electromagnetic wave equations and Euler equations. Electromagnetic wave simulations are used in the design of electrical machines (e.g., as indicated by Chari (1983)), modeling of antenna, radar, and satellites (e.g., as shown by Vandenbosch (2004)), defense systems (e.g., as demonstrated by Sankaran (2019)), medical imaging (e.g., as demonstrated by Lucano et al. (2016)), and oil and gas exploration (e.g., as shown by Abubakar et al. (2016)). On the other hand, the Euler equation also has many applications, including aerodynamics (as shown by Jameson (1983)) and aircraft/missile design (as demonstrated by Anandhanarayanan et al. (2013); Oktay et al. (1999).

Finally, I note that successful strategies for efficient computation of the acoustic and elastic wave equations can also be applied to the electromagnetic waves, as they share the same structural similarities. It also highlights the applicability and importance of this dissertation for broader applications.

2.3 Mathematical Foundation for Computation

This section briefly explains the mathematical terminology used in numerical methods in this dissertation. This includes the discretization method of discontinuous Galerkin (dG), the Gauss-Lobatto-Legendre spatial integration scheme, and the low-storage Runge-Kutta (LSRK) temporal integration method. While this section will not go into detail, it gives an overview of numerical methods for unfamiliar readers. Interested readers can find further reading materials in each subsection for more thorough explanations.

---

c For computing the solution for the next time-step, scalar values (i.e., derivative values and solution field from the current time-step) are combined to form updates, depending on the algorithms. During this step, communication between immediate neighboring elements is needed.

d The updates are combined with the solution values from the current time-step to advance the solution forward for the next time-step.

These operations are repeated for many time-steps as needed.

2.3.1 Discontinuous Galerkin Discretization

Three methods are commonly used for the numerical simulation of acoustic and elastic wave equations, which have been comprehensively discussed by Poursartip et al. (2020): Finite Difference Method (FDM), Spectral-Element Method (SEM), and discontinuous Galerkin Method (dG). All three methods enable efficient, explicit time-stepping, which helps parallel scalability.

In practice, FDM is the most commonly used method, typically for uniform grids that lead to fast Level-1 BLAS computations. However, it entails higher communication costs than SEM and dG, which becomes problematic for data exchange between compute nodes (Section 2.5.1) since deep ghost exchange is needed in FDM when high-order methods are used. In addition, it is difficult to develop stable high-order schemes for non-uniform finite-difference grids, which is needed to limit the dispersion error as shown by Ihlenburg (1998) (Appendix A.3.3).

On the other hand, SEM is typically used in regional seismology, where, due to the large size of the studied region, topography needs to be represented accurately, as described by Poursartip et al. (2017). Unstructured mesh generation could be very challenging and labor-intensive. Furthermore, efficient and accurate fluid-solid coupling is very challenging for SEM.

Finally, the dG method is a compact and robust finite element method that is locally conservative, stable, and high-order accurate as described by Cockburn et al. (2000); Grote et al. (2006); Wilcox et al. (2010); Hesthaven and Warburton (2010). As explained by Baggag et al. (2000), by using the dG method, the solutions are local to each finite element. Therefore, each element can be considered as an individual entity that needs to obtain some boundary data from its neighbors. Appendix A.3 briefly overviews the dG method.

The inter-element communication pattern for each method is shown in Figure 2.2. The dG method only needs information on the face of neighboring elements, SEM needs face and corner information on neighboring elements, and FDM needs

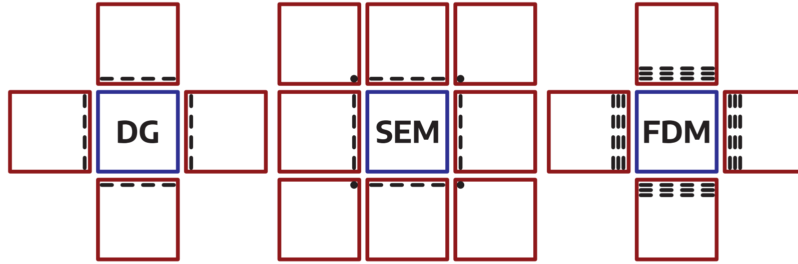


Figure 2.2: Comparison of inter-element communication pattern between DG, FDM, and SEM methods. DG only needs information from the face of neighboring elements; SEM needs information from the face and corner of neighboring elements; FDM needs information from the face and behind the face of neighboring elements.

the face and behind-the-face information on neighboring elements. The lower communication cost of dG compared to FDM and SEM makes it attractive to harness the computing power of modern and emerging architecture, especially where communication is the key bottleneck in many industry-relevant applications.

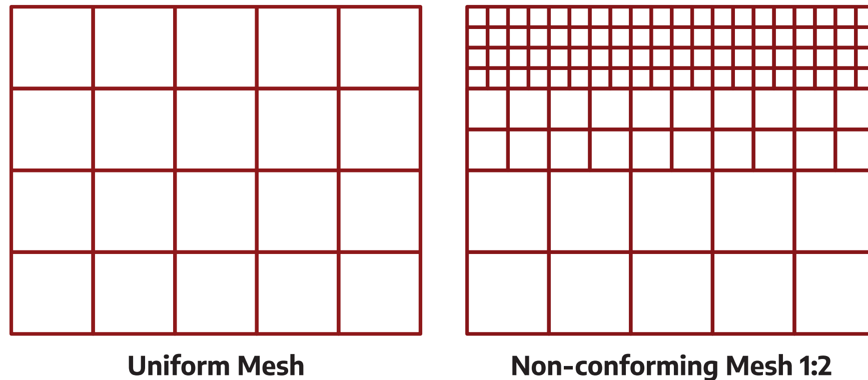


Figure 2.3: Comparison of uniform mesh and non-conforming mesh. While the work in this dissertation focuses on uniform meshes, most of the techniques can be applied to non-conforming meshes. Non-conforming mesh improves computational efficiency by adding higher resolution only when needed instead of having uniform high-resolution meshes.

While this dissertation focuses on uniform mesh, most of the proposed techniques can be extended to non-conforming mesh, as shown in Figure 2.3. The non-conforming mesh provides more computational efficiency by adding higher-resolution

meshes only when needed. In addition, robust and stable algorithms for dG discretizations on non-conforming meshes have been developed recently, as demonstrated by Kozdon and Wilcox (2018).

2.3.2 Gauss-Lobatto-Legendre Spatial Integration

There are several approaches to computing the spatial integrals on hexahedral elements using numerical quadrature, which includes Gauss-Legendre (GL) and Gauss-Lobatto-Legendre (GLL). The latter is the numerical quadrature used in this dissertation due to the reduced computational costs. However, there is a phenomenon called node clustering with GLL, which adds other challenges from a mathematical perspective. This section compares GL with GLL, highlighting the reason for choosing GLL as the numerical quadrature.

2.3.2.1 Gauss-Legendre (GL) Quadrature

Gauss-Legendre (GL) quadrature allows for accurate computing of the spatial integral, ensuring stable discrete formulation. This is particularly useful when a non-conforming mesh is used or when the variability of material within an element is present. However, with GL, the mass matrix is not diagonal, and thus, computing the inverse of the mass matrix is costly; it needs to be inverted at each local element involving multiplication with a solution vector. An N-point GL-quadrature has a complexity of $\mathcal{O}(N^6)$ and $\mathcal{O}(N^4)$ for volume and surface integrals, respectively, and can exactly integrate polynomials of order $2N-1$.

2.3.2.2 Gauss-Lobatto-Legendre (GLL) Quadrature

Gauss-Lobatto-Legendre collocates the quadrature nodes and the finite element nodes, decoupling the derivative computations in one direction from solution values in other directions. This is similar to how derivatives are computed with the finite difference method (FDM). In addition, using GLL results in the diagonal mass

matrix, where the inverse can be calculated easily, significantly reducing computational costs. However, this comes at the cost of the under-integration of high-order expressions. This leads to an unstable discrete formulation when a non-conforming mesh is used, or the variability of material within an element is present. Interested readers should consult the work by Kozdon and Wilcox (2018) for discretely stable dG formulation using GLL quadrature for non-conforming mesh and variable material properties. The computational complexity of an N-point GLL quadrature is $\mathcal{O}(N^4)$ and $\mathcal{O}(N^2)$ for volume and surface integrals, respectively, and exactly integrates polynomials of order $2N - 3$. This is significantly lower than using the GL quadrature.

2.3.2.3 Node Clustering Phenomenon with GLL

Nodes are clustered towards the sides of the elements when the nodes of an element and the associated quadrature scheme follow the GLL rule, as shown in Figure 2.4. The clustering phenomenon results in non-uniform spacing between nodes within an element. While the non-uniform spacing is necessary to prevent the Runge phenomenon, it creates two major challenges for seismic wave simulations.

First, it limits the allowable time step significantly, following $\Delta t \sim 1/p^2$, where $p = N - 1$ is the order of the polynomial. Second, it decreases the node density in the middle of the element, making it challenging to represent sharply varying material properties. For instance, for $N = 8$, node spacing in the middle of the element is 47% wider than the corresponding uniformly-spaced case.

Although it is beyond the scope of this dissertation, there are some possibilities to alleviate the node-clustering situation. Interested readers should consult the strategy proposed by Hesthaven and Warburton (2010) who use Gauss-Jacobi-Lobatto (GJL) nodes and Hale and Trefethen (2008); Kosloff and Tal-Ezer (1993) who transplants quadrature nodes and weights through a map function.

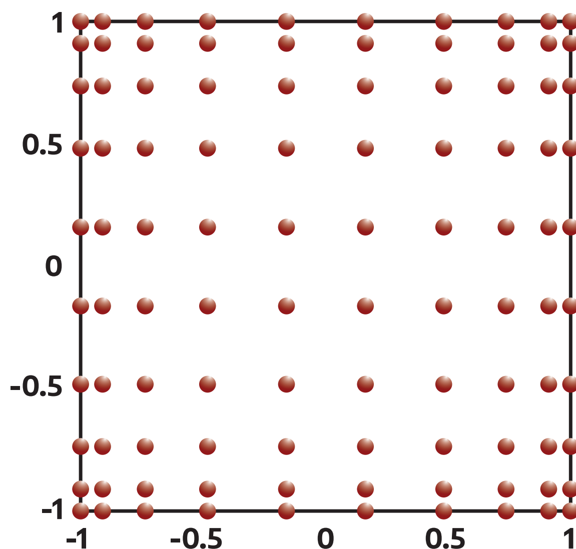


Figure 2.4: The node clustering phenomenon with GLL quadrature, showing an element with ninth-order polynomial in 2D space.

2.3.3 Fourth-Order Low-Storage Runge-Kutta Temporal Integration

A widely used method for temporal integration of time-dependent partial differential equations (PDEs) is the method of lines (MoL), as stated by van der Houwen (1996). The method of lines transforms PDEs into ordinary differential equations (ODEs) containing some spatial differential operators. However, not all ODE solvers are appropriate for the structure of the space-discretized PDEs. With the separation of time and space discretization, the Runge-Kutta method, a well-known stable ODE integrator, can be used for temporal integration. It numerically integrates the ODE by canceling out lower-order error terms using a trial step at the midpoint for the higher-order, as described by Butcher and Wanner (1996); Haelterman et al. (2009).

The Low-Storage Runge-Kutta is a version of the Runge-Kutta method that significantly reduces memory requirements while maintaining the versatility and robustness of standard Runge-Kutta. This property is attractive to modern computing architectures, such as GPUs, where the size of on-chip memory is limited. The work by Diehl et al. (2010) gives a detailed comparison of 75 LSRK methods, including

efficient second-order and third-order methods, when used with Maxwell equations discretized using dG.

This dissertation uses the fourth-order Low-Storage Runge-Kutta (LSRK4) for temporal integration. LSRK4 has five stages and needs only one extra storage array, as explained by Carpenter and Kennedy (1994). In contrast, the standard fourth-order Runge-Kutta method needs four extra storage arrays. While the detailed explanation of the Runge-Kutta will not be covered in this dissertation, interested readers should consult works by Williamson (1980); Niegemann et al. (2012); Ketcheson (2010) for a thorough explanation. Specific to dG discretization method, readers are recommended to refer to works by Kanevsky et al. (2007); Simonaho et al. (2012); Seny et al. (2014) .

2.4 Third-Party Software Libraries

Although the wave simulation application discussed in this dissertation uses many third-party libraries, two of them are the most important: the adaptive mesh refinement library and the message-passing interface library. This section looks at these two libraries to familiarize the readers since both are heavily discussed and impact the implementation of CPU (Chapter 4) and GPU codes (Chapter 5). On the other hand, the PIM codes (Chapter 7) do not rely on these libraries since all of the implementations are done manually, although they still follow the execution and data flows of both CPU and GPU codes.

2.4.1 Adaptive Mesh Refinement (AMR) Library

Adaptive mesh refinement (AMR) is a simulation technique that acts like a computational microscope, enabling researchers from various fields to focus on more intricate or scientifically important simulation regions. For instance, cosmologists can zoom in on cosmic filaments for deeper analysis, astrophysicists may target areas of nucleosynthesis, combustion scientists can study the complex chemistry near a flame

front, and seismologists can accurately handle high-frequency waves on complex material interfaces. This method greatly enhances the ability to examine computationally demanding problems in fine detail by directing computational resources primarily to the areas that need the highest accuracy. Interested readers are recommended to consult the book by Plewa et al. (2005).

The wave simulation investigated in this work utilizes `p4est` to manage the mesh of computational elements inside the problem domain. Although `p4est` can dynamically refine or coarse the mesh, in this work, the mesh is fixed once initialized, and thus it will not change during the simulation runtime. The load-balancing feature for partitioning and distributing mesh across different CPUs is an important part of `p4est`, although it does not support GPUs by default. Porting this feature to GPUs is one of the objectives of this dissertation. The documentation of p4est can be accessed in paper by Burstedde et al. (2011); Isaac et al. (2015, 2012)

2.4.2 Message Passing Interface (MPI) Library

In parallel computing, the message-passing model has emerged as an efficient and well-understood paradigm for parallel programming. However, in the early days, there was no standardization on syntax and semantics since every library had implemented its version. Most computer vendors have proprietary libraries that are not compatible with each other. Hence, developing and deploying applications on various systems and clusters is difficult.

As described by Gropp et al. (1996), standardizing the message-passing started in 1992 when the Message Passing Interface (MPI) Forum organized the workshop on Message Passing Standardization. The goal was to provide the standard on semantics, syntax, and low-level routines of the message-passing paradigm, allowing for wide portability. It is used as a communication protocol in distributed-memory and shared-memory multiprocessors and computing clusters independent of network settings and memory architecture, providing high performance, scalability, and porta-

bility, as described by Sur et al. (2006); Nielsen (2016).

2.4.2.1 Standard and Implementation

The first version of the standard, MPI 1.0, was finalized in 1994 by Message Passing Interface Forum (1994). It defines the basic point-to-point communication, collective communication, and data types. Subsequent versions clarified this first version: the MPI 1.1 released in 1995 by Message Passing Interface Forum (1995), the MPI 1.2 released in 1996 by Message Passing Interface Forum (1997), and the MPI 1.3 released in 1997 by Message Passing Interface Forum (2008a). The MPI 2.0 was released in 1997 by Message Passing Interface Forum (1997) and introduced parallel I/O, Remote Memory Access (RMA), dynamic processes, one-sided communication, extended collective communications, and external interfaces. This version was stable for ten years before MPI 2.1 was released in 2008 by Message Passing Interface Forum (2008b), and MPI 2.2 was released in 2009 by Message Passing Interface Forum (2009). These two standards provide minor updates and clarifications to the MPI 2.0 standard.

The MPI 3.0 standard was released in 2012 by Message Passing Interface Forum (2012) and introduced non-blocking collectives, new one-sided communication operations, and a unified RMA model. The subsequent standard, the MPI 3.1, was released in 2015 by Message Passing Interface Forum (2015), providing fixes and clarifications to its predecessor. The next major version of the MPI standard is MPI 4.0, which was released in 2021 by Message Passing Interface Forum (2021) and introduced persistent collectives, partitioned communications, improvements in error handling, and application assertions. The newest version is MPI 4.1, released in 2023 by Message Passing Interface Forum (2023), which provides clarification and fixes to its predecessor. Since the MPI 4.0 was released recently after the work on this dissertation began, the MPI 3.1 standard will be discussed instead.

Since the MPI standard only defines the syntax, semantics, and behavior of the

message-passing library, the implementations depend on the vendor of the MPI library without sacrificing the portability. Hence, there is not just one unique MPI library; many MPI libraries implement the MPI standard. They may perform differently across different systems, clusters, and network topologies. Some of the commonly-used implementation of MPI includes MPICH by Gropp et al. (1996), MVAPICH2 by Panda et al. (2021), SpectrumMPI by IBM Corporation (2024), OpenMPI by Gabriel et al. (2004), and Intel MPI by Intel Corporation (2024),

2.4.2.2 Communication Functions

The most commonly used communication pattern in MPI is point-to-point communication. It transfers the message from one specific sender MPI process to a particular receiver MPI process, both residing in the same communicator[2]. The sender must execute `MPI_Send` command while the receiver must execute `MPI_Recv` command. Through these commands, both must specify with whom they are communicating (i.e., source/destination of the message), the message identification (i.e., tag), and the type and size of the message. Point-to-point communication can be blocking or non-blocking. The former means that the process must wait until the message transmission achieves a particular state before executing the following instructions. The latter means that the process only issues the sending/receiving request and can continue its execution flow. The `MPI_Isend` and `MPI_Irecv` are the non-blocking version of `MPI_Send` and `MPI_Recv`, respectively.

Another type of communication pattern in MPI is collective communication. It involves communication between the members of an MPI communicator to manipulate a shared set of information. Collective communication is used for synchronization, data movement, and global computation. In synchronization, MPI provides `MPI_Barrier` to synchronize all processes within a communicator, which

[2]Communicator is a group of MPI processes that can communicate with each other. The default communicator that contains all processes is `MPI_COMM_WORLD`

will halt the execution flow of each process until all processes within a communicator reach and call the `MPI_Barrier`. In data movement, MPI provides broadcast `MPI_Bcast`, gather `MPI_Gather`, scatter `MPI_Scatter`, all-gather `MPI_Allgather`, and all-to-all `MPI_Alltoall` operations. Finally, in global computing, MPI provides reduce `MPI_Reduce`, all-reduce `MPI_Allreduce`, and scan `MPI_Scan` operations. While all these collective communication functions are blocking, the MPI 3.0 standard provides non-blocking collective communication functions.

Finally, MPI provides a one-sided communication pattern. It is an interface to Remote Memory Access (RMA), allowing single process to initiate communication activity on both sides (i.e., sender and receiver). This contrasts point-to-point communication where the sender and receiver must call `MPI_Send` and `MPI_Recv`, respectively. This is useful for irregular data transfer that follows some general pattern since it avoids the synchronization barriers, reducing overhead.

2.4.2.3 Remote Direct Memory Access (RDMA)

Remote Direct Memory Access (RDMA) extends the Direct Memory Access (DMA) technology by allowing a computer to access the memory of another computer connected through a network without the involvement of an operating system, processor, or cache. As described by Tekin et al. (2021); Potluri et al. (2013); Sharkawi and Chochia (2020); Venkatesh et al. (2014); Hamidouche et al. (2015), RDMA improves compute throughput and lowers communication latency as it frees many hardware resources. Operations such as read and write can be performed without interrupting the remote computer's CPU.

RDMA requires Network Interface Cards (NICs) and the networking standards that support RDMA protocol. The InfiniBand and the RoCE (RDMA-over-Converged-Ethernet) support RDMA protocol and are commonly used in high performance computing clusters. RoCE adds RDMA support to Ethernet since, in its basic form, Ethernet does not support Remote Direct Memory Access (RDMA), does

not guarantee the arrival of traffic packets (lossy), and does not guarantee the latency of traffic packets, as described by Kenny and Ulmer (2019); Tekin et al. (2021); Mittal et al. (2018). With RDMA, the NIC can directly transmit or receive the data from or to the CPU memory without explicitly staging the data in the NIC's buffer, intuitively called zero-copy.

2.4.2.4 CUDA-Aware Features

MPI with CUDA-Aware feature can distinguish memory buffers stored in CPU or GPU memory. It can directly send and receive memory buffers stored in GPU memory without the need to stage them first in CPU memory, significantly reducing the communication overhead. 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 available bus (e.g., NVLink). In contrast, 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.

2.5 Analysis of Computation and Communication

When accelerating applications to run on specific hardware, the relation between computation and communication is one of the most essential aspects to analyze. Optimizations often focus only on the computation part of the applications, while communication is frequently overlooked. In reality, data movement is the key bottleneck in modern computing systems, necessitating more attention to the communication part of the applications.

2.5.1 Identifying Data Movement

Although the DG-based discretization yields algorithms with higher data locality, data movement is still needed. Three types of data movement inside the dG-based wave simulation applications must be considered, as explained below.

- **Intra-Device Data Movement.** CPUs and GPUs have limited on-chip memory, and thus, they rely on off-chip memory to store a large portion of the data. Fetching data from off-chip memory is expensive in terms of energy and performance, and thus, data movement must be orchestrated carefully.

- **Inter-Device, Intra-Node Data Movement.** As the problem grows, the off-chip memory capacity of a single CPU or GPU can no longer hold the necessary data. Adding more devices is a viable option to handle the larger problems by partitioning and distributing them across devices. However, each device must synchronize the data at some point, creating another type of data movement that could potentially impact the overall simulation performance.

- **Inter-Device, Inter-Node Data Movement.** The size of the problem that can be handled by single compute node is limited by how many CPUs or GPUs are installed. With increasingly large problems, using multiple compute nodes becomes necessary. This adds another type of data movement since each compute node must exchange the data through the inter-node communication network, which is often the weakest link in computer systems, potentially limiting the overall simulation performance.

2.5.2 Roofline Analysis

As described by Williams et al. (2009); Hanindhito and John (2024), the roofline chart visualizes the performance of workloads or kernels, comparing them with the computing capabilities of the hardware with which they run. This visualiza-

tion gives insight into whether the workloads/kernels are compute-bound or memory-bound, helping devise optimization plans.

The roofline chart is a log-log plot where the y-axis represents the compute throughput measured in GFLOP/s, and the x-axis represents the arithmetic intensity measured in FLOP/byte. The compute throughput indicates the number of floating-point or integer operations performed per second. In contrast, the arithmetic intensity suggests the amount of computation (i.e., FLOP/s) that can be done per data byte. Figure 2.5 shows an example of roofline chart with AMD Instict MI300X GPU model.

The roofline model, plotted into the roofline chart as the roof and slope lines, models the computing hardware (e.g., CPU, GPU, and other hardware accelerators). The roof represents the peak compute throughput the hardware can perform on specific data types (i.e., arithmetic precision). The model can have multiple roofs, accommodating different precisions and functional units to perform the arithmetic operations. Meanwhile, the slope represents the peak memory bandwidth for a specific memory type. Although the model can have multiple slopes accommodating each memory type in the hierarchy (e.g., first-level cache, last-level cache, DRAM), often, the off-chip memory (DRAM) is the one that is drawn on the plot since it is the critical bottleneck.

The hardware model can be obtained theoretically from the manufacturer's datasheet or whitepaper. The hardware model in Figure 2.5 is obtained from the whitepaper by Advanced Micro Devices (2023). The hardware model can also be obtained empirically through measurements. Empirical Roofline Toolkit by Yang (2015) provides a framework to obtain the roofline model of computing hardware through measurements.

The position of kernels or workloads on the roofline chart can be determined by profiling them. Using the profiling tool provided by the manufacturer, metrics such as the number of operations, the number of memory read and write, and the execution duration can be measured, which helps plot kernels or workloads on the

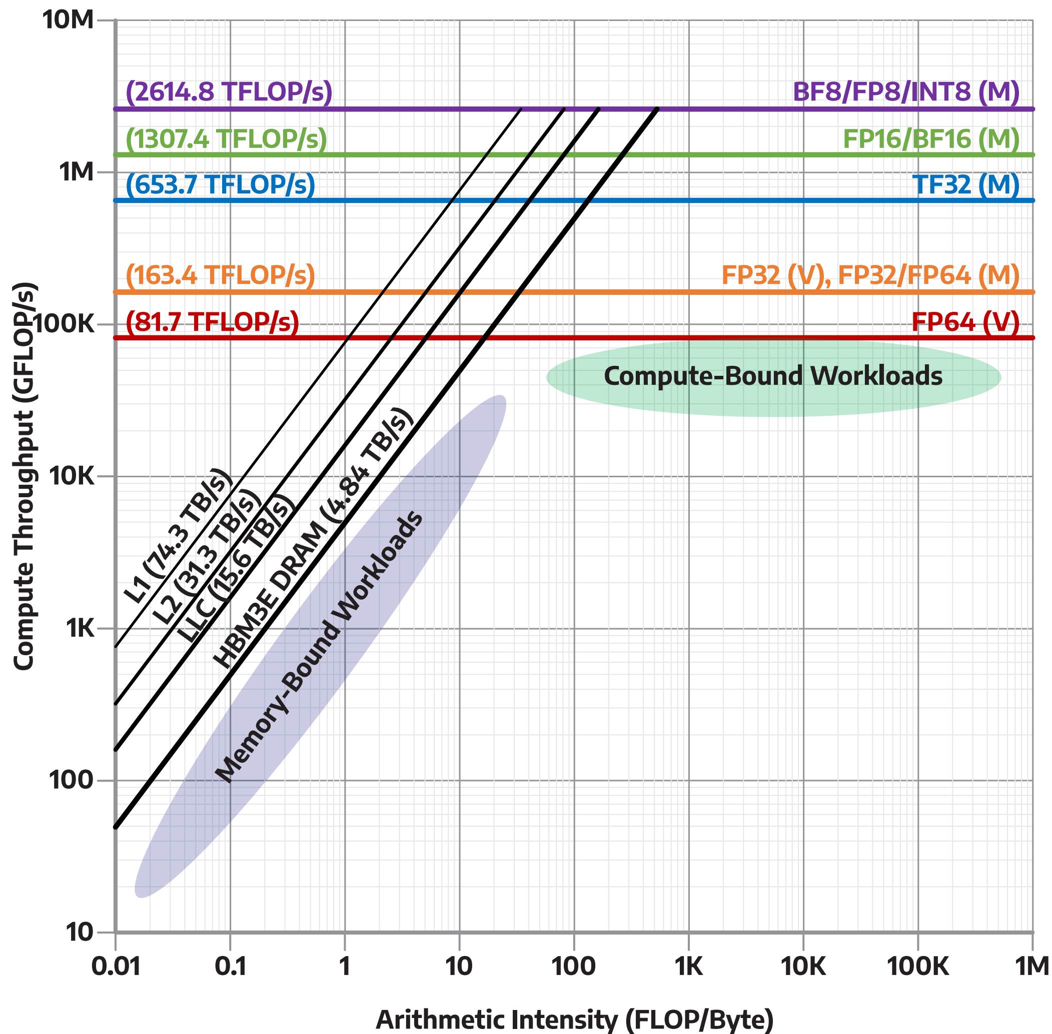


Figure 2.5: Example of roofline chart for analyzing compute-communication relation, showing the AMD Instinct MI300X GPU model. The compute throughput for various arithmetic precisions and the bandwidth for each memory hierarchy are derived from the whitepaper published by Advanced Micro Devices (2023). The area with the blue shadow is the location of memory-bound workloads, while the area with the green shadow is the position of compute-bound workloads. The **V** indicates vector (SIMD) units while the **M** indicates matrix units.

roofline chart. Based on the position, it can be determined whether the kernels or workloads are compute-bound (i.e., closer to the roof of the hardware model) or memory-bound (i.e., closer to the slope of the hardware model). Then, optimization strategies can be determined and applied according to this characterization.

2.6 Graphics Processing Unit (GPU) and Its Applications

Graphics Processing Unit (GPU) is the popular hardware accelerator for highly parallel applications. Although historically, GPU is solely used to accelerate graphics applications, GPU has become a general-purpose accelerator for many applications, including high-performance scientific applications and machine learning, as described by Hanindhito et al. (2024). Featuring a massive number of ALUs hierarchically grouped, GPU is suitable for executing applications that have regular flow, predictable memory access patterns, and abundant parallel operations.

The discretization of the problem domain (Section 2.3.1) yields many smaller discrete elements that can be processed in parallel. In addition, inside each element, the unknown variables can be evaluated in many discrete points in parallel. These two sources of parallelism are why wave simulations have abundant inherent parallelism. A computing platform that can extract as much parallelism as possible for this type of workload is desired, as it can shorten the time needed to calculate the solutions. GPU is among the hardware candidates that can efficiently accelerate wave simulations, as explored in Chapters 5 and 6. Therefore, this section provides a brief introduction to GPU to the readers since having some familiarity with it will help them understand the subsequent chapters.

2.6.1 Road to Become General-purpose Massively-Parallel Accelerator

Before the 2000s, GPUs were fixed-function accelerators solely to process graphics applications. Since most operations in graphics applications are highly paral-

lel[3], GPUs have been designed as massively parallel processors to extract these parallelisms as explained by Blythe (2008). In the early 2000s, GPUs became increasingly programmable due to the need to create more complex computer-generated imagery, as described by Blythe (2008); Elliott (2004). However, the programmability is limited to graphics functions, such as pixel and vertex shaders, through API such as Direct3D and OpenGL.

In 2006, the introduction of Tesla architecture by NVIDIA as described by Lindholm et al. (2008), along with its CUDA programming interface explained by Buck (2007b), marked the significant shift of graphics processing unit (GPU) from fixed-function graphics accelerators to general-purpose accelerators as discussed by Fatahalian and Houston (2008); Harris (2008); Buck (2007a). In summary, GPUs were fixed-function accelerators (before the 2000s), became programmable with programmable shaders (early 2000s), easier to program with unified shaders (early 2006), and became general-purpose massively-parallel accelerators (2007). Nowadays, GPU has become a vital hardware accelerator for applications that have abundant inherent parallelism as shown in the work by Nickolls et al. (2008b,a); Che et al. (2008), including General Matrix Multiplication (GEMM), as described by Sorokin et al. (2022); Li et al. (2018b); Abdelfattah et al. (2017); Brown et al. (2020).

2.6.2 Hardware and Software Perspective

Before developing applications on GPU, it is essential to understand how the functional units are organized and how the programming models work. It is also important to be familiar with how the hierarchical organization in software is mapped into the hierarchical organization of hardware inside the GPU. Figure 2.6 shows the hierarchy of GPU from software and hardware perspective and how they relate.

[3] i.e., primitives, fragments, and pixels can be processed in parallel during each stage of the graphics pipeline.

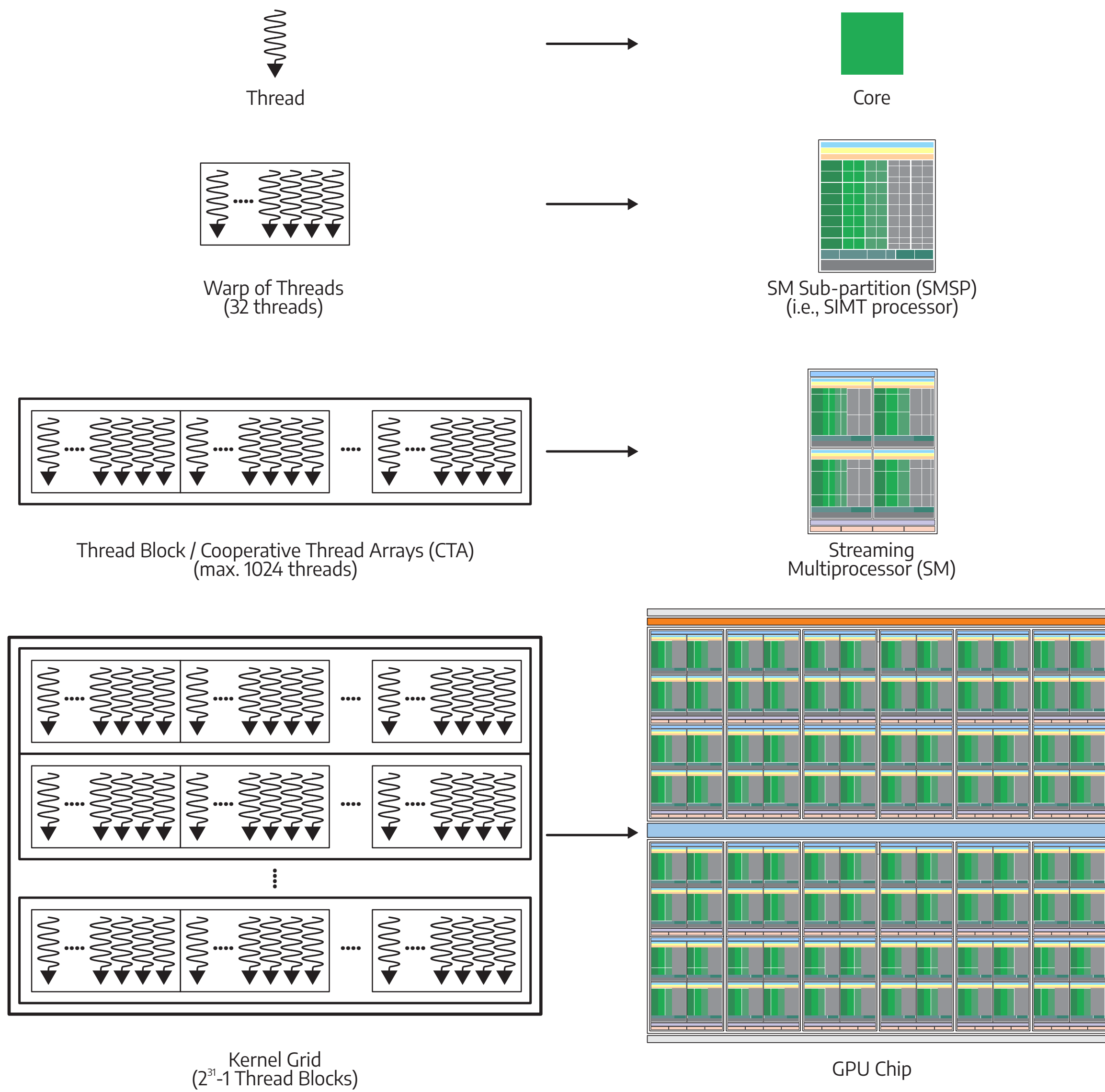


Figure 2.6: The organization of GPU, as viewed from a software perspective (left) and hardware perspective (right), and how they relate to each other hierarchically. A thread is executed by a "core"; A warp of threads is mapped into SM Subpartition; A thread block is scheduled into a Streaming Multiprocessor; and a grid is launched to a GPU chip.

2.6.2.1 Terminology

Since this dissertation uses NVIDIA GPU to accelerate the wave simulation as discussed in Chapter 5, the explanation here uses terminology from NVIDIA. Although other GPU vendors, such as AMD and Intel, use different terminology, they use similar organizations. The terminology equivalency for GPU across different vendors is given in Table 2.1.

2.6.2.2 Hardware Perspective

The manufacturers often advertise their GPUs as having thousands of cores[4] to signify their massively parallel architecture. However, the term *cores* in GPUs is not the same as in CPUs; it refers to the execution units (i.e., ALUs) instead of the fully-fledged CPU core. These GPU *cores* are grouped into one processor[5], which looks more like a CPU core with significantly many ALUs. In NVIDIA GPU, this processor is called a Streaming Multiprocessor (SM). There can be 100s of SMs inside a GPU chip, each with many *cores* (ALUs), for a total of thousands of *cores*. A simplified diagram of an SM inside NVIDIA Volta is given in Figure 2.7, as described by Choquette et al. (2018).

Newer NVIDIA GPUs may have slightly different SM architecture to improve performance and add new features. Interested readers should consult the whitepaper released by NVIDIA to find detailed SM architecture for each generation of NVIDIA GPUs: Volta by NVIDIA Corporation (2017), Turing by NVIDIA Corporation (2018), Ampere by NVIDIA Corporation (2020a), Hopper by NVIDIA Corporation (2022), and Ada Lovelace by NVIDIA Corporation (2023b).

SM comprises four SM sub-partitions (SMSP) and several types of on-chip memory, which include L1 instruction cache, L1 data cache, shared memory, and tex-

[4] CUDA Cores (CC) in NVIDIA GPUs, Stream Processors (SP) in AMD GPUs, or Vector Engines (XVE) in Intel GPUs.

[5] Streaming Multiprocessor (SM) in NVIDIA GPUs, Compute Unit (CU) in AMD GPUs, Compute Slice (SLC) in Intel GPUs.

<table>
<tr><th>NVIDIA</th><th>AMD</th><th>Intel</th><th>Description</th></tr>
<tr><td colspan="4">Software Perspective</td></tr>
<tr><td>Thread</td><td>Work-item</td><td>Work-item</td><td>Single stream of instruction and data.</td></tr>
<tr><td>Warp</td><td>Wavefront</td><td>Sub-group</td><td>Group of threads that execute the same instruction stream in lock-step fashion and operate on different data.</td></tr>
<tr><td>Thread Block</td><td>Work-group</td><td>Work-group</td><td>Group of warps executed by single SM/CU/XC and share synchronization barrier and on-chip memory.</td></tr>
<tr><td>Grid</td><td>ND-Range</td><td>ND-Range</td><td>Collection of thread-block or work-group of a kernel executed by GPU.</td></tr>
<tr><td colspan="4">Hardware Perspective</td></tr>
<tr><td>CUDA Cores (CC)</td><td>Stream Processors (SP)</td><td>Vector Engines (XVE)</td><td>Arithmetic unit that processes one data item and executes portion of SIMT instruction stream.</td></tr>
<tr><td>Tensor Cores (TC)</td><td>Matrix Cores</td><td>Matrix Engines (XMX)</td><td>Specialized unit to accelerate matrix-matrix operations (e.g., GEMM).</td></tr>
<tr><td>Subpartition (SMSP)</td><td>SIMD Unit</td><td>Execution Unit (EU)</td><td>SIMT processor to execute warp/wavefront/sub-group in a lock-step manner.</td></tr>
<tr><td>Streaming Multi-processor (SM)</td><td>Compute Unit (CU)</td><td>Xe Core (XC)</td><td>Unit capable of executing one thread-block/work-group of a kernel. It consists of subpartitions sharing on-chip memory.</td></tr>
<tr><td>Texture Processing Cluster (TPC)</td><td>Workgroup Processor (WGP)</td><td rowspan="2">Render Slice or Compute Slice (SLC)</td><td>GPU super-clusters, consisting of several SMs/CUs/XCs to perform texture operations.</td></tr>
<tr><td>Graphics Processing Cluster (GPC)</td><td>Shader Engine</td><td>GPU mega-clusters, consisting of several super-clusters to perform graphics operations.</td></tr>
<tr><td>Graphics Processing Die (GPD)</td><td>Graphics Complex Die (GCD)</td><td>Stack (STK)</td><td>GPU dies, consisting of several megaclusters; used for chiplet-based GPU.</td></tr>
</table>

Table 2.1: The terminology equivalency between NVIDIA, AMD, and Intel GPUs. Note that the Graphics Processing Die (GPD) is an unofficial terminology given for completeness.

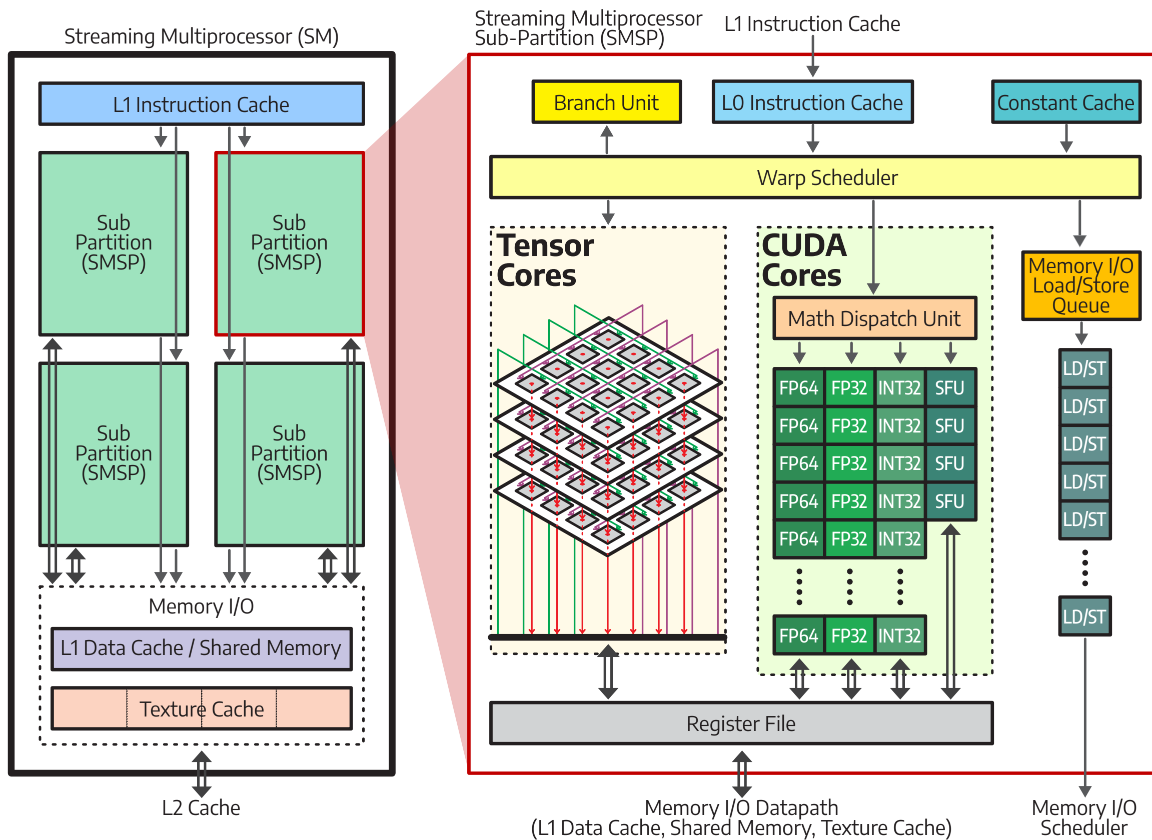


Figure 2.7: A simplified diagram of the Streaming Multiprocessor inside NVIDIA Volta GPU. Each SM contains four SM Subpartitions sharing L1 instruction cache, L1 data cache, shared memory, and texture cache. There are CUDA Cores, Tensor Cores, register files, L0 caches, and warp schedulers within the subpartition.

ture cache. The organization of the on-chip data memory (i.e., L1 data cache, texture cache, and shared memory) has changed multiple times across different architectures of NVIDIA GPUs. To give the programmers flexibility on how on-chip data memory should be managed, starting from Volta architecture, the L1 data cache, texture cache, and shared memory are implemented as unified on-chip memory. Programmers can choose to either allocate 100% of on-chip data memory as a cache managed by the hardware or dedicate some portions of it as shared memory managed by the programmers. Programmers who use shared memory must carefully manage its usage since the reduced amount of L1 and texture caches can degrade the performance. Finally, the L2 cache is the last-level cache for Volta and comprises many slices shared across all SMs. It interfaces directly with the off-chip memory (e.g., GDDR or HBM). Section 2.6.3 explains the memory hierarchy of the GPU in more detail.

Diving deeper into the SMSP, there are several on-chip memory: L0 instruction cache, constant cache, and register files. CUDA Cores (CC) are the default computation units, consisting of FP64 (double-precision floating-point ALUs) units, FP32 (single-precision floating-point ALUs) units, INT32 (integer ALUs), and Special Function Units (SFUs). The number of FP32 units is usually used to advertise the number of CUDA Cores in GPUs. The SFUs compute the transcendental functions, such as trigonometric functions. Datacenter class GPU features significantly more FP64 units to handle high-performance computing applications that use double-precision floating-point arithmetic for accuracy-sensitive computation. Finally, specialized units called Tensor Cores are added to Volta and newer generation GPUs to accelerate General Matrix Multiplications (GEMMs), which are abundant in many machine learning workloads. Tensor Cores will be briefly discussed in Section 2.6.4.

2.6.2.3 Software Perspective

As described by Lindholm et al. (2008), GPU uses Single-Instruction Multiple-Thread (SIMT) execution model, which is a modification to Single-Instruction Multiple-Data (SIMD). In addition to executing one instruction with multiple data (SIMD),

SIMT applies one instruction to multiple independent threads in parallel. This allows programmers to write data-parallel code for coordinated threads and thread-parallel code for individual threads.

GPU kernel is a small program that runs on GPU. A kernel can have millions of threads, collectively called a grid. Inside a grid, the threads are further grouped into thread blocks or cooperative thread arrays (CTAs). A grid can have as many as $2^{31} - 1$ thread blocks, each containing up to 1024 threads. Thus, theoretically, a kernel can have as many as 2 trillion threads[6]!

The global scheduler, GigaThread Engine in NVIDIA GPU, schedules each thread block into the SM. It also manages the context switches of the thread blocks in each SM. Ideally, multiple thread blocks are scheduled into each SM, allowing for aggressive context switching; when one thread block stalls (e.g., due to memory access), it can run another thread block to hide the latency and keep the SM busy.

Finally, each warp inside a thread block is scheduled into an SMSP. The warp scheduler inside the SMSP maps the threads within the warp to the cores, which run the execution in a lock-step fashion. Any differences in the execution path (e.g., due to different branch outcomes) within the warp will cause thread divergence. Due to the divergence, instead of running in parallel, the threads within the warp will run serially based on their execution path. Note that thread divergence only occurs within the warp since each warp can be independently executed.

2.6.3 Memory Hierarchies

With thousands of cores (ALUs), the memory subsystem of the GPU has complex tasks to feed data into all cores, keeping the SMs busy. With thousands of threads running concurrently, the register files of the GPU are significantly larger than

[6] i.e., $(2^{31} - 1) \times 1024 = 2,199,023,254,528$

the CPU[7]. Modern datacenter-class GPU uses 3D-Stacked High-Bandwidth Memory to provide terabytes of bandwidth into the GPU chip. However, even with this complex memory organization, the applications that run on GPU are often memory-bounded. Like the compute parts of the GPU, the memory parts have two views: the logical view from the software perspective and the physical view from the hardware perspective. Both are explained as follows.

2.6.3.1 Logical View

The logical view of the memory hierarchy of the GPU is used when developing the applications. It includes the memory scope among the threads, warps, and blocks, the allocation lifetime, and the data type that can be stored. Figure 2.8 illustrates the logical view of GPU memory related to the organization of GPU from a software perspective. Below are the types of GPU memory from a logical perspective.

- **Registers**. This is the fastest memory to store immediate operands and intermediate results for each thread. Each thread has some register allocations determined during the compile time. Although the compiler allocates it, users can limit the allocation for each thread using decorator `__launch_bounds__`.
- **Local Memory**. Each thread uses the local memory as private memory to store temporary variables and operands. It is used when there are register spills due to insufficient registers to store intermediate data. It has a limited scope for each thread and is not visible to other threads. The hardware manages local memory.
- **Shared Memory**. Shared memory is a user-managed memory used for threads to share data. However, it is only visible to threads within the same thread

[7] As shown in work by Larabel (2023), Intel Sapphire Rapids CPU has 42,752 bytes of register in each core for a total of 2,505 KB of registers in 60-core variants. On the other hand, the NVIDIA H100 GPU has 256 kB of register per SM, for a total of 33 MB of registers across 132 SMs, as shown by NVIDIA Corporation (2022).

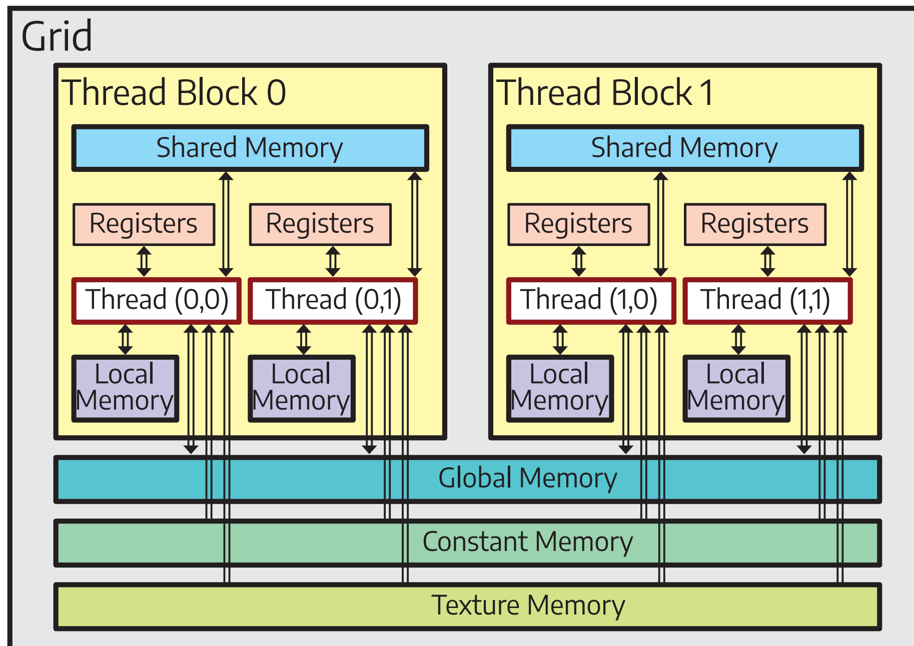


Figure 2.8: The logical view of GPU memory hierarchy as seen by the programmers. The global, constant, and texture memory are visible to all threads within the kernel grid. Global memory is read/write memory, while constant and texture memories are read-only. A shared memory is used for fast data sharing between threads inside a thread block. Each thread has allocated private registers and local memory, which stores operands and intermediate results when register spillage occurs.

block. Reading and writing from shared memory is significantly faster than going to global memory. In Volta or newer GPUs, shared memory must be activated before it can be used, as it is part of the unified on-chip data memory.

- **Global Memory**. As the name suggests, the global memory is visible to all threads inside the kernel grid. Users are responsible for allocating the global memory and managing its read and write access.

- **Texture Memory**. This is a specialized memory for storing 2D read-only texture data. It is visible to all threads inside the kernel grid. Users are responsible for initializing the texture memory before launching the kernel since the contents inside are read-only from the kernel.

- **Constant Memory**. This is a specialized memory for storing 1D constant data. It is visible to all threads inside the kernel grid. Like the texture memory, users are responsible for initializing the constant memory before launching the kernel since the contents inside are read-only from the kernel.

2.6.3.2 Physical View

The physical view of the memory hierarchy is the actual memory hierarchy implemented on the hardware of the GPU. As shown in Figure 2.9, it consists of registers, L1 data cache, shared memory, texture cache, L2 data cache, the off-chip device memory, and the host memory for NVIDIA Tesla V100, as an example. Note that, for simplicity, some of the on-chip memory is omitted from the figure, which includes the L0/L1 instruction caches and constant cache as shown in Figure 2.7. The closer to the cores (ALUs), the lower the access latency and the higher the memory bandwidth, and thus, when developing applications, users must consider storing repeatedly-used data on lower-level memory. Below are the types of GPU memory from the physical view.

- **Registers**. Registers are the fastest memory directly interfacing with the cores (ALUs). It is used to store immediate operands and intermediate results. With a large number of threads, each SMSP contains relatively large registers. This large register size is also used to support GPU aggressive context switching, where the context for each thread block handled by the SM is kept in registers. However, register availability is often insufficient for handling 1024 threads per thread block, especially for kernels with many intermediate results.

- **Constant Cache**. As the name suggests, the constant cache stores the constant data stored in constant memory. It provides high-speed access to constant data.

- **Shared Memory**. Shared memory is user-managed scratchpad memory that stores repeatedly used data and shares data between threads within a thread block. In Volta or newer generations on GPU, shared memory is implemented as unified on-chip memory, which shares the same structure as the L1 and texture caches. Using shared memory will reduce the capacity of the L1 cache, and thus, it must be done carefully to avoid performance degradation.

- **L1 Data Cache**. L1 cache is hardware-managed memory that stores frequently used data within an SM. It is implemented as unified on-chip memory, which shares the same structure as shared memory and texture cache. By default, the unified on-chip memory is used for the L1 and texture data cache.

- **Texture Cache**. As the name suggests, the texture cache stores the 2D constant texture data stored in texture memory.

- **L2 Cache**. L2 cache is the last-level cache for many NVIDIA GPUs. It stores the data evicted from the L1 cache and is shared across all SMs. It interfaces directly with the off-chip memory.

- **Device Memory**. Device memory is the off-chip DRAM memory implemented as GDDR or HBM. This is the largest capacity of memory that the GPU has,

but it is the slowest one. Accessing the data from and to this memory incurs significant latency, and the limited bandwidth is more likely to be the bottleneck of GPU performance.

- **Host Memory**. Host memory is the DRAM attached to the host CPU. Before executing the kernels, the data from the host memory is often copied to the GPU memory through the PCIe bus, which is slower. After the kernel executes, the results are copied back to the host memory. The data movement between host and device memory can be managed manually by the programmers or automatically by the hardware and the driver. However, excessive data movement between the host and device memory will significantly reduce the GPU performance.

2.6.4 The Inclusion of Matrix Accelerator

CUDA Cores are the workhorse that executes operations in parallel; thus, NVIDIA GPUs have thousands of them. In 2017, NVIDIA added specialized units called Tensor Cores into their GPUs with the launch of Volta architecture, as described by NVIDIA Corporation (2017). Tensor Core provides a significant performance boost and energy efficiency when performing General Matrix Multiplications (GEMMs), which is abundant in many HPC and AI/ML workloads.

As shown in Figure 2.7, Tensor Cores comprise ALUs organized in a 3D systolic array structure. The inclusion of FP64 precision into third- and fourth-generation Tensor Cores opened the possibility of using them for accelerating HPC applications that require higher accuracy, as shown in the work by Lee et al. (2022); Gallet and Gowanlock (2022). Other manufacturers followed by integrating matrix accelerators into their GPUs: AMD with Matrix Cores, as described by Advanced Micro Devices (2020), and Intel with XMX Matrix Engine, as described by Jiang (2022).

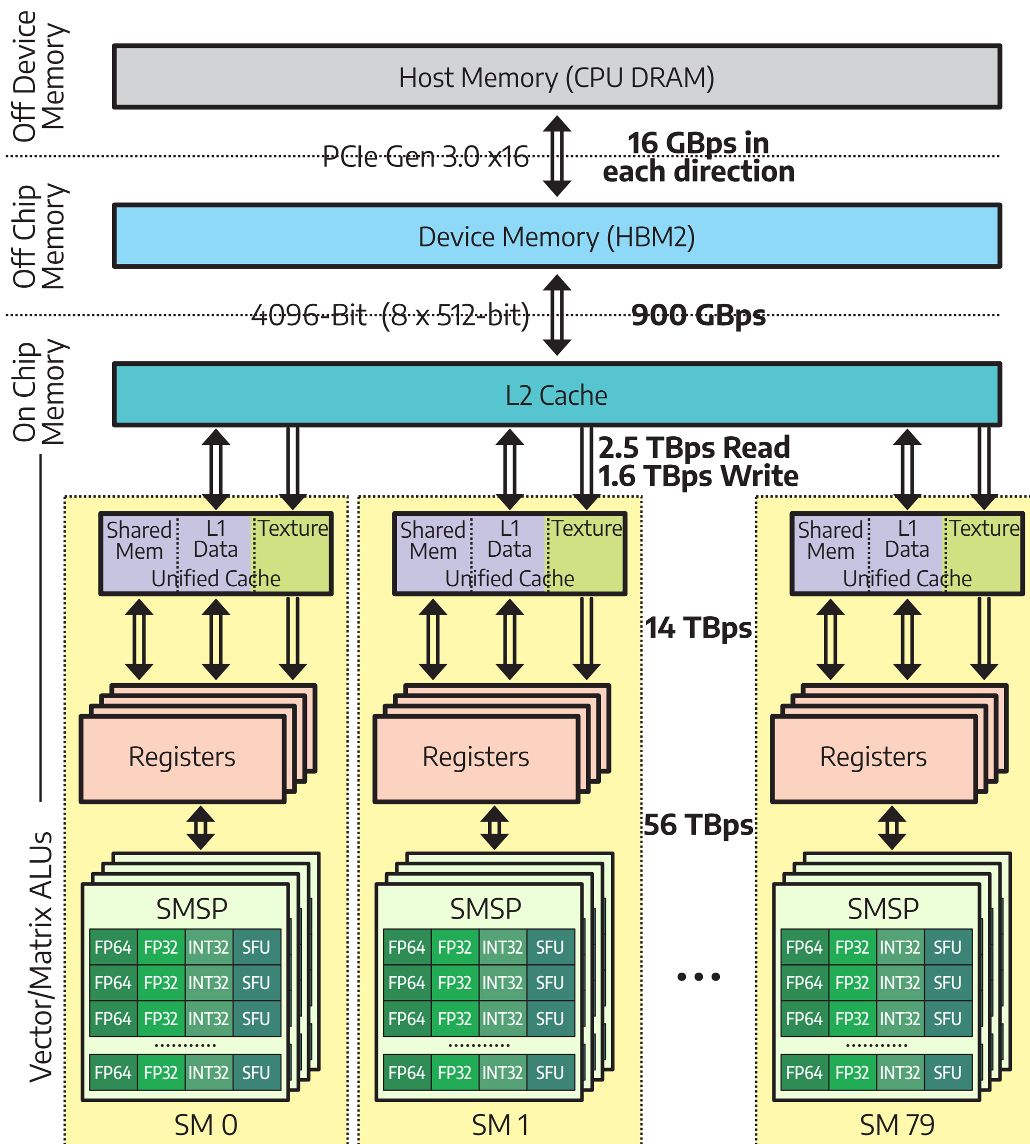


Figure 2.9: The physical view of memory hierarchy in NVIDIA Tesla V100 GPU. The registers are the fastest memory with limited capacity, interfacing directly with the cores (ALUs) and providing more than 56 TBps aggregate bandwidth. The unified on-chip memory implements the shared memory, L1 data cache, and texture cache, providing fast access to repeatedly-used data to each SM. The L2 cache is the last-level cache shared by all SM and interfaces directly with the off-chip device memory. The device memory is the largest capacity but the slowest memory, with only 900 GBps bandwidth. Finally, the host memory is the memory attached to the host CPU and may be used as the slowest buffer in case the device memory is insufficient.

2.6.5 Mixed-Precision Computation

Mixed precision computation, such as mixed precision training in machine learning as described by Micikevicius et al. (2018), can help reduce the amount of memory required to perform the computation, ease the bandwidth requirement (e.g., off-chip memory and inter-node network bandwidth), and lower the computational power needed. It employs different precision formats: lower precision (e.g., half-precision, such as FP16) and higher precision (e.g., single-precision, such as FP32). Lower precision is used for parts of the computation that should not significantly impact numerical accuracy. On the other hand, higher precision is used for critical parts of the calculation to ensure numerical stability and accuracy.

Some of the hardware has single-precision arithmetic units that can execute half-precision arithmetic twice the rate of FP32, such as NVIDIA Pascal architecture, as described by NVIDIA Corporation (2016), which improves training performance. The matrix accelerator, briefly discussed in Section 2.6.4, supports lower precision data type with very high computational throughput than the vector units. Given the numerous benefits of mixed precision training, companies continuously seek more efficient data formats. Google developed BF16, which preserves the dynamic range of FP32 in a 16-bit format as described by Wang and Kanwar (2019). NVIDIA also introduced TF32, as described by Choquette et al. (2021), which maintains the dynamic range of FP32 but offers the accuracy of FP16 in a 19-bit format.

2.6.6 Previous Work on GPU-Accelerated Scientific Applications

This section highlights several related works that use the dG method on GPUs. The work by Abdi et al. (2019) tried to solve three-dimensional Euler equations that govern the thermodynamic state and the atmosphere's motion. GPU acceleration is essential to obtain more accurate results within a given simulation time limit. They used Nvidia Tesla K20x GPU to accelerate their computation and achieved 15x speed-up over 16-core AMD Opteron 6274. They extended their work to support multiple

GPUs for up to 16,384 GPUs with a scaling efficiency of 90%.

Another effort to accelerate computation based on the dG method was made by Chan et al. (2016). They tried to use GPU to accelerate the dG method problem for hybrid meshes. Their hybrid meshes contain vertex-mapped hexahedral, wedge, pyramidal, and tetrahedral elements. They also presented energy-stable discrete formulations for the hexahedron using Gauss-Legendre and Gauss-Legendre-Lobatto nodal bases. They did not compare the speed-up they obtained using GPU to the CPU implementation. Instead, they presented the computational efficiency of the solver in terms of floating-point operation per second and estimated bandwidth when running in single Nvidia GTX980 GPU in single precision.

The work by Gandham et al. (2015) used GPU to accelerate discontinuous Galerkin methods for solving shallow water equations for modeling tsunamis, storm surges, and tidal waves. They use OpenCL to map nodal DG discretization into GPU for both AMD (AMD Radeon HD7970) and Nvidia (Nvidia Tesla C2050) GPU, which consists of volume kernel, surface kernel, and update kernel. Depending on the polynomial order, the GPU's time-step efficiency is 8x higher than the Intel Core i7-3930K CPU.

Moreover, the work by Karakus et al. (2019) used GPU to accelerate the discontinuous Galerkin method for an incompressible flow solver. The equations are unsteady incompressible Navier-Stokes, which are discretized in time using a semi-implicit scheme consisting of implicit treatment of the split Stokes operator and explicit treatment of the nonlinear term. They optimized the performance of the most time-consuming kernel by tuning bandwidth usage, memory utilization, and fine-grain parallelism. They did not compare the GPU implementation to the CPU implementation. Interestingly, they presented a roofline model for the achieved floating-point performance for various implementations of the GPU kernel.

Work by Mu et al. (2013) did one of the closest works to ours where arbitrary high-order discontinuous Galerkin (ADER-DG) method for solving 3D elastic seismic

wave was successfully ported to GPU on unstructured tetrahedral meshes. They reached a speed-up of 24.3x for the single-precision and 12.8x for the double-precision on the Nvidia Tesla C2075 GPU compared to the single-core version of CPU code running on Intel Xeon W5880.

Moreover, the work by Modave et al. (2016) tried to analyze GPU performance on acoustic and elastic models using a nodal discontinuous Galerkin method. They did not compare the speed-up to the CPU version. They analyzed and compared three different GPU kernels in terms of net arithmetic throughput. This was obtained by dividing the total number of FLOP per time-step by the run-time required for one time-step update.

Alternatively, the work done by Heinecke et al. (2019) uses Many Integrated Core (MIC) hardware instead of GPU. They used Intel Knights Mill CPU and Intel Knights Landing CPU to accelerate fused discontinuous Galerkin simulation. They compare the performance of Intel Xeon Phi 7250 processor (Knights Landing), Intel Xeon Phi 7295 processor (Knights Mill), and dual Intel Xeon Platinum 8180 (Skylake) in terms of non-zero peak efficiency. The Skylake reaches the highest non-zero peak efficiency for single-precision computation, followed by Knights Landing and Knights Mill. The Knights Mill reaches the highest non-zero peak efficiency for the double-precision workload, followed by Skylake and Knights Landing.

Lastly, the work by Karakus et al. (2016) discussed level set reinitialization for an adaptive discontinuous Galerkin method. It used OCAA as an abstract programming model to encapsulate native languages for CUDA, OpenCL, Pthreads, and OpenMP. There are three significant computations for finding the solution: volume integrals, surface integrals, and time-step updates. Each of them is implemented on separate GPU kernels. They compare the performance on each kernel between Nvidia Tesla C2075 GPU using OpenCL-compiled kernel and CUDA-compiled kernel and Intel Xeon E5-2670 using OpenMP-compiled kernel. In single-precision computation, the OpenCL-compiled kernel and CUDA-compiled kernel achieve speed-ups

of up to 8x and 6x for the volume kernel and up to 18x and 20x for the surface kernel compared to the OpenMP-compiled kernel.

2.6.7 Previous Work on GPU-Accelerated dG-based Discretization

GPUs have been used to accelerate many scientific applications from diverse domains. Therefore, the related works presented here are not an exhaustive list. In fact, these works show how important the GPUs are for scientific computation as they can potentially speed up the computation. Works by Wang et al. (2019b,c); Shu et al. (2020); Cheng et al. (2020); Ren et al. (2018); Xu et al. (2017) use GPU to accelerate lattice Boltzmann solver. GPU is also used for accelerating the modeling of smoothed particle hydrodynamics of granular flow by Chen et al. (2020), elliptic problems solver on unstructured hexahedral meshes by Remacle et al. (2016), and genetic algorithms by Cheng and Gen (2019).

In addition, the acceleration benefits from GPUs are also helpful for calculating time integration of the shallow water equations on the sphere as shown by Archibald et al. (2015), simulating red blood cells as conducted by Blumers et al. (2017), and modeling finite volume coastal ocean as done by Zhao et al. (2017). Furthermore, computational diffusion MRI by Hernandez-Fernandez et al. (2019), asteroid shape modeling by Engels et al. (2019), channeling radiation of relativistic particles simulation by Nielsen (2019), and Monte-Carlo simulation by Wei and Kruis (2013) benefit from acceleration provided by GPUs. Specific to DG, the work by Wolf et al. (2022) developed an efficient time-stepping scheme for seismic waves in poroelastic media, providing the community with open-source code called SeisSol, a scientific software for numerical simulation of seismic waves in either CPU and GPU.

Although the following works use FPGA, they are worth mentioning. The works by Gourounas et al. (2023a,b) develop the FPGA implementation of dG-based wave simulation. In large-scale scenarios, the work by Faj et al. (2023) develops a dG-based shallow-water model on unstructured mesh running on multi-FPGAs.

2.7 Processing-in-Memory (PIM) and Its Applications

The von Neumann architecture, which constitutes most modern computing systems, has separate centralized memory and compute units. Due to the limited bandwidth, the interface between the compute units and the off-chip memory often becomes the bottleneck in overall system performance, especially for memory-bound applications that operate on large datasets. The data movement between the compute units and the off-chip memory consumes much energy, which becomes more concerning. For instance, shown in the works by Keckler et al. (2011); Kestor et al. (2013), moving 256-bit of data by 10 mm consumes more energy than performing a floating-point arithmetic operation in double precision.

Compute-in-memory attempts to address the bottleneck with von Neumann architecture and reduce the energy consumption due to the data movement by bringing the compute units closer to where the data is stored, which is the memory, as shown by Khoram et al. (2017); Mutlu et al. (2019). They have demonstrated promising performance with reduced energy consumption of compute-in-memory technologies, making them attractive for data-intensive applications. However, as an emerging technology, the broad adoption of compute-in-memory relies on the availability of better software stacks (e.g., compiler, library, framework), helping users migrate their existing codes to leverage these technologies, as described by Ghose et al. (2019)

There are two computing-in-memory approaches: Near-Memory-Processing (NMP) and Processing-in-Memory (PIM). NMP integrates the compute units near the memory arrays at the chip or package level. The amount of computation that can be performed depends on how much area is allocated to the compute units. If compute units are integrated at the chip (die) level as the memory arrays, there will be competition for space, which typically results in simpler compute units. PIM, on the other hand, performs the computations directly in the memory arrays. Depending on the memory technology, it usually requires minimal changes to the memory array structures. It also requires altering the memory commands issued by the memory

controller to perform the computations. Different memory technologies have been investigated to be used for PIM: SRAM (e.g., as demonstrated by Eckert et al. (2018)), DRAM (e.g., as described by Gao et al. (2019); Seshadri et al. (2017)), and non-volatile memory technologies, which include phase-change memory (e.g., investigated by Hoffer et al. (2022)), resistive RAM (e.g., as described by Imani et al. (2019a); Hanindhito et al. (2021)), spintronic RAM (e.g., as explained by Chowdhury et al. (2018)), and NAND Flash (e.g., as described by Gao et al. (2021)).

This dissertation uses PIM technology to accelerate wave simulation, which will be discussed in Chapter 7. Although the latency of the arithmetic operations in PIM may be higher as CMOS-based designs due to the bit-by-bit NOR operations, PIM chip can run many parallel operations inside each memory block. For example, the work by Imani et al. (2019a) shows that 8 million parallel operations can be achieved on a 1 GB PIM chip. Based on its operation, PIM can be divided into two categories: analog PIM (e.g., as explained by Feinberg et al. (2018); Cheng et al. (2017); Cai et al. (2018)) and digital PIM (e.g., as described by Imani et al. (2019a); Kvatinsky et al. (2014); Siemon et al. (2015)). Both analog and digital characteristics of non-volatile memory have been investigated by Shafiee et al. (2016); Song et al. (2017); Zhang et al. (2020b) to support arithmetic operations in memory.

2.7.1 Analog PIM

Analog PIM leverages inherent electrical properties of the memory arrays to perform computations according to Kirchoff's law. For example, it modulates analog input signals into weighted analog output signals, as shown by Shafiee et al. (2016); Song et al. (2017); Zhang et al. (2020b) or performs dot product operations by applying different voltages into each word-line and bit-line. The drawback of analog-based PIM is its reliance on digital-to-analog (DAC) and analog-to-digital (ADC) converters. Both of these components consume most of the chip area and power, especially for SRAM-based and NVM-based analog PIMs, as shown by Talati et al. (2016).

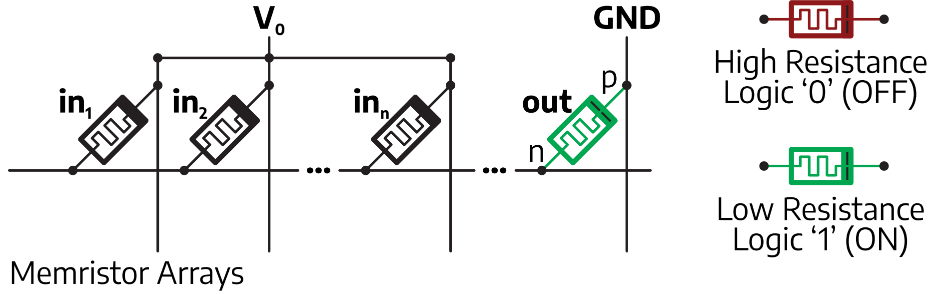
In addition, analog circuits are more sensitive to noise, manufacturing variations, temperature changes, and voltage fluctuations.

2.7.2 Digital PIM

Digital PIM eliminates the ADC/DAC issue present in Analog PIM by exploring the characteristics of the crossbar circuits, as shown by Talati et al. (2016). It performs basic digital logic operations, such as NOR, on the memory arrays. For example, in resistive-based digital PIM built using memristors, each memristor cell's resistance can change between ON and OFF depending on the voltage level applied to the bit-line and word-line. The resistance behavior of the memristor is used to represent a logic '1' and '0'. The output of the memristor is initialized as ON (i.e., logic '1'). When one or more inputs switch from logic '0' to '1', the output of the memristor will switch from ON to OFF (i.e., logic '0'), implementing the NOR operation.

Figure 2.10 illustrates the bitwise NOR operations in memristor-based PIM, which is also explained in work by Imani et al. (2019a). The memristor has two states: OFF, where it has high internal resistance, and ON, where it has low internal resistance. The output memristor can switch from ON to OFF when the voltage across the memristor (i.e., between the p-terminal and n-terminal) exceeds a certain threshold, as demonstrated by Kvatinsky et al. (2015). The output memristor is initialized to ON at the beginning to implement the NOR operation. Then, the execution voltage, V_0, is applied at the p-terminal of input memristors while the p-terminal of the output memristor is connected to the ground. If one or more input memristors have logic 1 (i.e., low resistance, ON), the output memristor will switch state from ON (low resistance) to OFF (high resistance), representing logic 0. While Figure 2.10 shows the operation of PIM in a row-parallel way, it can also operate in a column-parallel way by transposing the word lines and the bit lines, as demonstrated by Talati et al. (2016).

More complex arithmetic operations, such as addition and multiplications, can



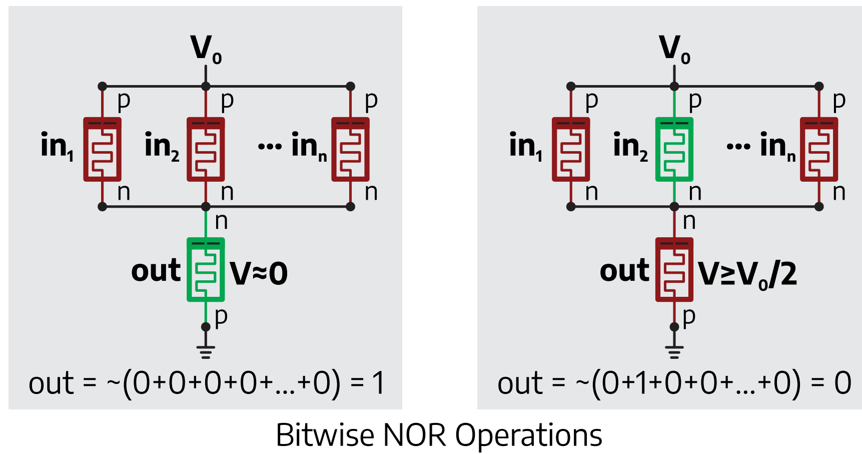


Figure 2.10: The NOR operations are the fundamental operations for memristor-based PIM with which more complex arithmetic operations are constructed. The output memristor is initialized to ON, and the execution voltage V_0 is applied at the beginning of the computation. If one or more input memristors are ON (logic 1), the output memristor will switch state to OFF (logic 0).

Fixed Point	$(6.5N^2 - 7.5N - 2) \times T_{NOR}$
Floating Point	$(12N_e + 6.5{N_m}^2 - 7.5N_m - 2) \times T_{NOR}$

Table 2.2: Multiplication latency for fixed-point and floating-point arithmetic in digital PIM. N represents the number of bits in fixed point numbers while N_e and N_m represent the exponential and mantissa bits in floating-point numbers. The T_{NOR} is the single bitwise NOR operation latency.

be represented through sequences of NOR operations. However, since the operands are processed in a bit-serial way, arithmetic operations have significantly longer latency, as shown in Table 2.2. Fortunately, the massively parallel operations that memory arrays of digital PIM can perform, coupled with reduced data movement between off-chip and on-chip memory, outweigh the longer latency for performing arithmetic operations, resulting in a performance advantage over the conventional von Neumann architectures.

2.7.3 Applications Leveraging PIM

Previous studies explored PIM for various applications, especially those that deal with a large amount of data, as shown by Imani et al. (2018, 2019b). A work by Chi et al. (2016); Zhang et al. (2020b); Song et al. (2017); Shafiee et al. (2016); Imani et al. (2019a) proposed a novel PIM architecture to accelerate neural network applications. The use of PIM in neural networks is also explored by Angizi et al. (2019), who compared both analog- and digital-based PIM. Other studies by the following authors explore PIM for accelerating graph applications: Dai et al. (2019); Ahn et al. (2015a); Chen et al. (2022). Other applications include PIM for clustering algorithms, as shown by Imani et al. (2020), PIM for blockchain, as indicated by Wang et al. (2020), and PIM for visualization systems, as demonstrated by Li et al. (2020b). However, limited work explores PIM in HPC and scientific computing applications. For example, the work by Asifuzzaman et al. (2023) investigated the advantage of PIM in HPC kernels. In addition, none of the works explore dG-based wave simulation on PIM.

Chapter 3: Methodology

This chapter explicitly explains how the research in this dissertation was conducted. It describes the methods, the hardware platform, the software libraries, and the tools used during the study. First, the CPU codes for wave simulation and how these codes are transformed into GPU and PIM codes are explained (Section 3.1). It describes the origin of the CPU code, the effort to transform and optimize the codes to run on the target hardware, and the strategy to verify the results. Next, the hardware platform where the experiments run is discussed in detail (Section 3.2). This includes the computing resources provided by the Texas Advanced Computing Center (TACC), the Maverick2 and the Longhorn clusters, and the purpose-built desktop.

Next, since there is no actual hardware available, the experiment with PIM is done by simulating the PIM model on cycle-accurate PIM simulation (Section 3.3). This includes a brief explanation of the cycle-accurate PIM simulator, the non-volatile memory models, the energy models, the PIM hardware configurations, and the process node scaling. Finally, the computing libraries and measurement tools are described (Section 3.4), which includes the tools to obtain performance and energy from the real hardware and the libraries used to compile and run the wave simulations.

3.1 Wave Simulation Code Base

The research collaborators gave the wave simulation codes investigated in this dissertation. The code base is proprietary and closed-source. However, for the research, I have access to the high-level language of the source codes. It is written in C++ and utilizes `Cmake` build system to generate the `makefile`, which is then used to compile the wave simulation applications, using either Intel compiler or GNU compiler. The codes have several third-party libraries as their dependencies. The two most notable ones are the `p4est` as adaptive mesh refinement library (Section 2.4.1)

and the message-passing interface library (Section 2.4.2). The application can run large-scale acoustic and elastic wave simulations using these two libraries across multiple compute nodes and processors. However, the codes do not support acceleration using GPU or PIM; they can only run on the CPU, and hence, they are called CPU codes throughout this dissertation. Accelerating the wave simulations using GPU and PIM are the main objectives of this dissertation.

3.1.1 CPU Codes Examination

The first step of the research is to investigate the CPU codes profoundly. This includes examining the data structures, the simulation flows, and the data flows. In addition, the third-party libraries are also discussed since they are tightly coupled to the CPU codes; the application has many function calls to these libraries. While the MPI has support for GPU acceleration (Section 2.4.2.4), the `p4est` does not. Therefore, plans must be made to bring GPU support to `p4est` for developing the GPU implementation, as discussed in Chapter 5. For PIM implementation discussed in Chapter 7, it does not deal with these third-party libraries since most stuff in PIM is done manually, and no compiler is available yet. In addition, there is no plan to support multi-chip and multi-node on PIM; thus, there is no need for a message-passing library. The investigation of the wave simulation CPU codes is briefly discussed in Chapter 4 with a deep dive in Appendix B.

3.1.2 Porting and Optimizing Codes

After thoroughly studying the CPU codes, the GPU implementation can be developed by adapting the data flow and simulation flow, discussed in Section 4.3. The approach for developing the basic GPU implementation is to use most of the code base and perform modifications to make it more suitable to run on GPU, extracting as much parallelism as possible. The mesh structure and elements are still initialized on the CPU, using `p4est` functions, and are copied to the GPU memory. After that,

it follows the simulation flow as the CPU code, except, that instead of launching the simulation kernel on the CPU, the kernel is launched on the GPU. Some modifications are done to the GPU kernels, and serial iterations over the elements are replaced by parallel executions of all elements. Some of the `p4est` functions are no longer relevant for GPU execution, and thus, they are replaced with strategies to achieve the same functions but are more GPU-friendly. In addition, multi-GPU support is added by adding methods to run the inter-GPU data exchanges and synchronizations. These are not straightforward processes and need meticulous efforts to successfully port the CPU code to run on GPU.

Developing the basic GPU codes is one thing; running the codes to take as much performance out of the GPU is another thing. The basic GPU codes are characterized using profiling tools by collecting hardware metrics. The characterization is important to investigate their behavior when being executed on GPU and identify the critical performance bottlenecks. Then, hardware-aware optimization strategies are planned and applied to develop more optimized GPU codes. Optimizations are done in steps, where the profiling is always done before moving to the next step to ensure the codes' updated behavior is considered when applying subsequent optimizations.

Developing PIM codes takes a different approach since there is no compiler to transform high-level language into low-level machine language. While the PIM codes still follow the CPU codes' simulation and data flow, all the work is done manually, from storing the data into the memory cells, generating instructions to run the execution, and managing the data movement within the PIM chip. The kernel execution flows receive some modifications to adapt to the row-parallel execution of PIM, extracting as much parallelism as possible. As with the GPU code, optimization techniques are developed after characterizing the basic PIM code, aiming to get more performance out of the PIM hardware. While multi-PIM runs are not considered, a technique is developed to let the PIM chip handle larger problem sizes that were previously impossible due to insufficient memory. Other techniques are also developed to increase the parallelism and the throughput of wave simulation running on PIM.

3.1.3 Results Verification

While having better performance and energy efficiency over the CPU codes are the primary targets for developing GPU and PIM implementations, the accuracy of the results is also essential. At the end of the simulations, the CPU code outputs the L_2 error and the ranges of the computed solutions as part of diagnostic and reporting shown in Figures B.3 and B.4. The GPU codes also have diagnostic and reporting by copying the simulation results from GPU memory to CPU memory and running the same functions on the CPU to calculate the L_2 error and the ranges of solutions. Comparing these values can ensure that CPU and GPU codes have the same functionality and results when running for a long simulation time. With the optimization of the GPU code done in steps, the result is also verified when any changes are made to the GPU code to ensure any problems that lead to incorrect results can be identified and solved as early as possible.

More thorough verification is done to compare each numerical value generated at the end of the simulations, as shown in Figure 3.1. First, the CPU codes run for several steps, and the numerical data is dumped into the disk as binary reference data. Then, the GPU codes run using the same configurations as the CPU codes and generate numerical data. At the end of the simulation, the GPU data is copied to the CPU memory, and using the binary reference data, all of the respective values are compared. Doing an exact comparison of floating-point numbers is impossible since floating-point operations are not commutative when done on computers. Instead, a very small number ϵ is defined as an acceptable range of differences between the values obtained from CPU and GPU codes.

Since the actual hardware is unavailable for PIM codes, the cycle-accurate PIM simulator is used (Section 3.3.1). This cycle-accurate simulator does not perform operations on the actual data; instead, it only gives the accurate cycles of all the operations as if they are performed on the actual data. Therefore, comparing the result at the end of the simulations, either through L_2 error, range of computed

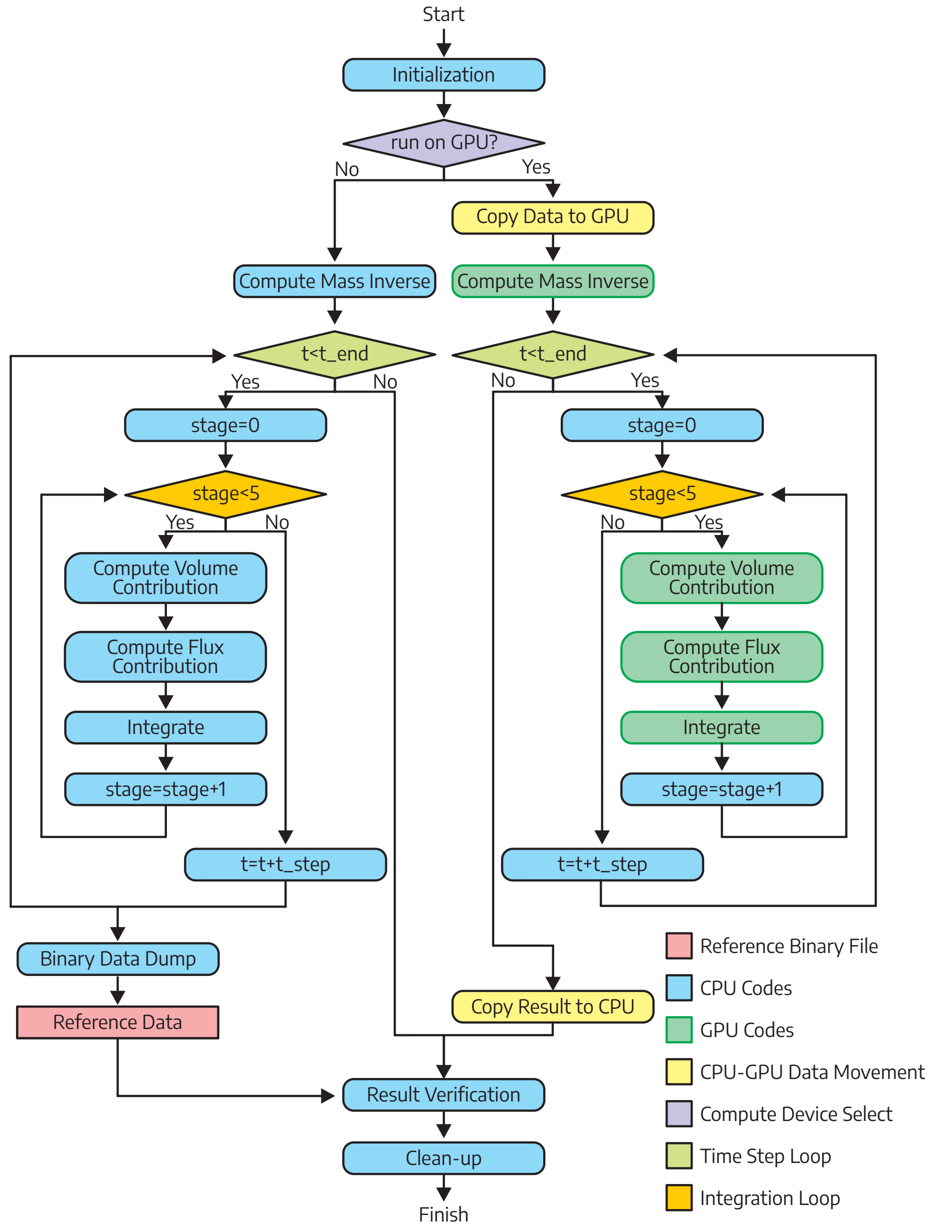


Figure 3.1: High-level execution diagram of the wave simulations to compare the results from the GPU codes with the CPU code for verification. First, the CPU codes run for several time steps, and at the end of the time steps, the binary data is dumped as a reference into the disk. Then, the GPU codes run for the same time steps, and the results are verified against the reference binary data.

solutions, or thorough values comparison, is impossible. The verification is limited to ensuring the simulation and data flow are performed accurately with no operations missed. It is a tedious process since all of the efforts of porting the CPU codes to PIM codes are done manually.

3.2 Computing Platform and Infrastructure

Various high-performance computing clusters are the backbone of the research conducted in this dissertation. The Texas Advanced Computing Center (TACC) at The University of Texas at Austin provides the computing clusters used in this dissertation. These computing clusters are equipped with graphics processing units (GPUs) and are interconnected through the InfiniBand network, making it possible to run realistic-scale wave simulations. However, since the research spanned more than six years, there have been changes in the availability of computing clusters at TACC. Thus, some parts of the dissertation use older computing clusters while others use newer ones. The Table 3.1 briefly summarizes the hardware configuration of each TACC computing cluster used in this dissertation. Other computing resources mentioned in Section 3.2.4 are also used for the experiment. The single-precision peak throughput for each GPU is obtained from TechPowerUp GPU Database: NVIDIA GeForce GTX 1080 Ti from TechPowerUp (2017a), NVIDIA Tesla P100 PCIe from TechPowerUp (2016), NVIDIA Tesla V100 16 GB PCIe from TechPowerUp (2017b), NVIDIA Tesla V100 16 GB SXM2 from TechPowerUp (2017c), and NVIDIA A100 40 GB from TechPowerUp (2020).

3.2.1 TACC Maverick2 Cluster

TACC Maverick2 was a cluster supporting GPU accelerated Machine Learning and Deep Learning research tasks. Each compute node has multiple GPUs, making it ideal for workloads that benefit a dense GPU cluster with minimal CPU usage. There are three types of compute nodes in Maverick2: `gtx`, `p100`, and `v100`. Texas

Platform	Maverick2			Longhorn	Lonestar6
Host CPU					
Model (# Sockets)	Intel Xeon E5-2620 v4 (2)	Intel Xeon Platinum 8160 (2)	Intel Xeon Platinum 8160 (2)	IBM POWER9 (2)	AMD EPYC 7763 (2)
Base / Turbo Clock	2.10 GHz / 3.00 GHz	2.10 GHz / 3.70 GHz	2.10 GHz / 3.70 GHz	2.10 GHz / 3.70 GHz	2.45 GHz / 3.5 GHz
Total Cores / Threads	16 / 16	48 / 48	48 / 48	40 / 160	128 / 128
Inter-socket Link	QPI 32 GB/s	UPI 41.6 GB/s	UPI 41.6 GB/s	X-Bus 64 GBps	IFIS 72 GBps
Accelerator (GPU)					
Model	GTX 1080 Ti	Tesla P100	Tesla V100	Tesla V100	A100
# GPUs per node	4	2	2	4	3
Form Factor	PCIe	PCIe	PCIe	SXM2	PCIe
Process Node	16 nm	16 nm	12 nm	12 nm	7 nm
Clock Frequency	1,530 MHz	1,480 MHz	1,582 MHz	1,582 MHz	1,410 MHz
Register Size	7,168 KB	14,336 KB	20,480 KB	20,480 KB	27,648 KB
L2 Cache Size	2,816 KB	4,096 KB	6,144 KB	6,144 KB	40,960 KB
Memory Size and Type	11 GB GDDR5X	16 GB HBM2	16 GB HBM2	16 GB HBM2	40 GB HBM2
Memory BW	484 GBps	720 GBps	900 GBps	900 GBps	1555 GBps
# FP32 CUDA Cores	3,584	3,584	5,120	5,120	6,912
FP32 Peak Throughput	11.3 TFLOPS	10.6 TFLOPS	15.7 TFLOPS	15.7 TFLOPS	19.5 TFLOPS
CPU-to-GPU Link	PCIe 3.0	PCIe 3.0	PCIe 3.0	NVLink 2.0	PCIe 4.0
GPU-to-GPU Link	PCIe 3.0	PCIe 3.0	PCIe 3.0	NVLink 2.0	PCIe 4.0
Computation Nodes					
# Nodes (# GPUs)	2 (8)	2 (4)	2 (4)	32 (128)	6 (18)
Inter-node Link	FDR MT27500	FDR MT27500	FDR MT27500	EDR MT28800	HDR MT28908

Table 3.1: The compute platform (node) configurations on computing clusters Maverick2, Longhorn, and Lonestar6 of Texas Advanced Computing Center (TACC).

Advanced Computing Center (2020b) provides the user guide for Maverick2. At the end of May 2023, Maverick2 was decommissioned.

There were 24 `gtx` nodes, each equipped with four NVIDIA GeForce GTX 1080 Ti 11 GB PCIe GPUs, two Intel Xeon E5-2620 v4 CPUs, and 128 GB of RAM. Each node was housed in a SuperMicro X10DRG-Q chassis. In addition, there were three `p100` nodes, each equipped with two NVIDIA Tesla P100 16 GB PCIe GPUs, two Intel Xeon Platinum 8160 CPUs, and 192 GB of RAM. Each node was housed in a Dell PowerEdge R740 chassis. Finally, there were four `v100` nodes, each equipped with two NVIDIA Tesla V100 16 GB PCIe GPUs, two Intel Xeon Platinum 8160 CPUs, and 192 GB of RAM. Each node was housed in a Dell PowerEdge R740 chassis. Each node was equipped with single Mellanox FDR InfiniBand MT27500 ConnectX-3 network interface card, capable of providing 56 Gbps unidirectional bandwidth.

Although there are many compute nodes, the user can only request two nodes, limiting the multi-GPU runs. Therefore, TACC Maverick2 is only used for developing, characterizing, profiling, and optimizing the single GPU implementation (Sections 5.3 and 5.4). In addition, the baseline for evaluating the processing-in-memory (Section 7.4) comes from the performance and energy number of single GPU run in Maverick2. Since only one GPU is utilized, the intra-node interconnect topology for each compute node and the inter-node interconnect topology for the whole cluster are not essential to discuss. TACC only disclosed Fat-Tree as the topology of inter-node interconnect without specifying how the nodes were connected.

3.2.2 TACC Longhorn Cluster

TACC Longhorn was a GPU cluster developed with IBM to handle GPU-accelerated workloads. Each node has four GPUs, making it well-suited for complex tasks requiring high GPU density and minimal CPU usage. The GPUs are data-center class GPUs capable of supporting scientific workloads that require double-precision floating-point arithmetic and machine learning workloads that require half-precision

(mixed) arithmetic. There are 96 standard nodes and eight large-memory nodes configured identically, except for the double memory capacity for the latter. Texas Advanced Computing Center (2020a) provides the user guide for Longhorn. At the end of November 2022, Longhorn was decommissioned.

Longhorn was the primary cluster used for multi-node multi-GPU experiments, allowing up to 32 nodes and 128 GPUs (Section 5.6.4). The compute node is IBM Power System AC922 with two IBM POWER9 CPUs, four NVIDIA Tesla V100 16 GB SXM2 GPUs, and two Mellanox EDR InfiniBand MT28800 ConnectX-5 EX network interface cards. Each CPU has 128 GB of eight-channel DDR4 in the standard node and 256 GB of eight-channel DDR4 in the large-memory node. Figure 3.2 shows the intra-node topology of IBM Power System AC922, as described by Nohria et al. (2018). Note that the CPU has `ppc64le` ISA, which is different than `x86-64`.

Since Longhorn was used for multi-node and multi-GPU experiments, it is crucial to understand what the intra-node interconnect topology looks like since it may affect the performance number, mainly where communication between GPUs exists. Each CPU is connected to two NVIDIA Tesla V100 GPUs, each using three sublinks providing 150 GBps bidirectional bandwidth. NVLink is also used to connect a pair of GPUs, providing 150 GBps bidirectional bandwidth between them. However, communication with GPUs on different pairs must traverse through the X-Bus inter-socket link, providing less than half bandwidth than NVlink does (i.e., 64 GBps). Each CPU has a dedicated InfiniBand EDR ConnectX-5 EX network interface card capable of offering 100 Gbps inter-node bandwidth.

For inter-node interconnect, TACC did not disclose the topology details; it only mentioned spine-leaf topology. Figure 3.3 approximates the topology of the Longhorn cluster. For simplicity, identical Mellanox SB7800 36-port InfiniBand EDR switches are used for leaf and spine switches. Each rack has one leaf switch, connecting each compute node within a rack using two downlinks. With 12 compute nodes per rack and two connections per compute node, the leaf switches handle 24 downlinks. Each

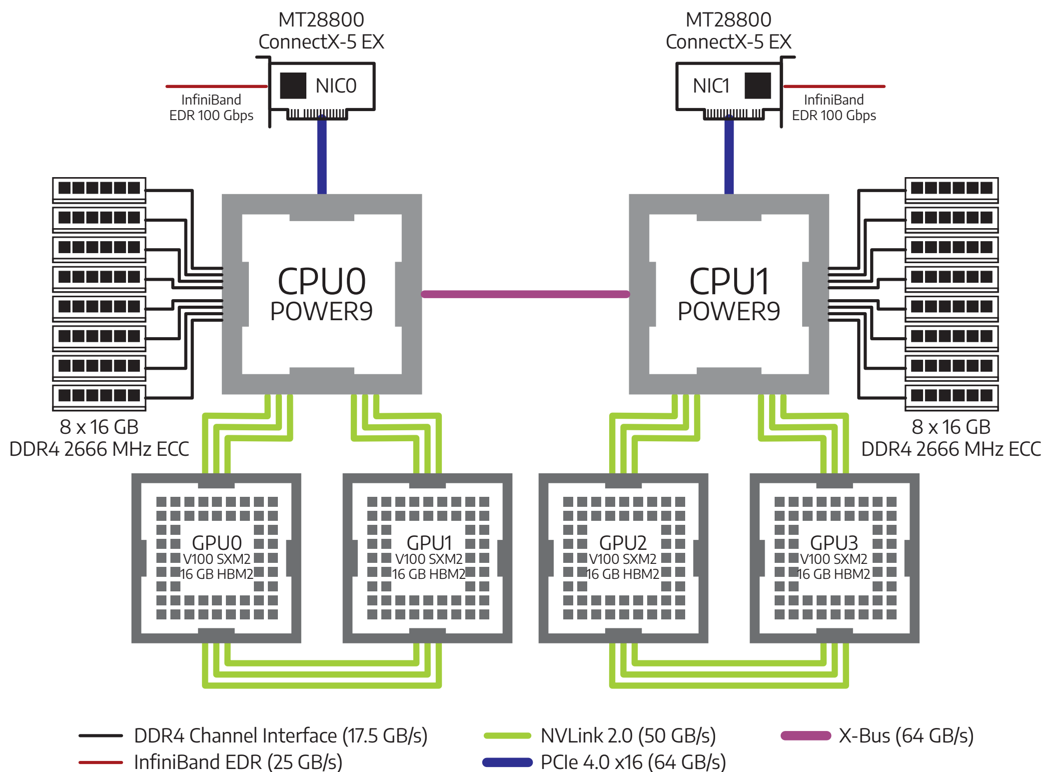


Figure 3.2: A simplified diagram of the interconnect topology inside each compute node on TACC Longhorn. Two IBM POWER9 CPUs, each equipped with eight-channel DDR4-2666. Both are connected through an X-Bus inter-socket link. Two NVIDIA Tesla V100 GPUs are connected to one CPU and are connected to each other via NVLink 2.0. Each CPU has a dedicated ConnectX-5 EX InfiniBand EDR Network Interface Card. Bandwidth numbers shown are bidirectional, except for half-duplex interfaces like DRAM.

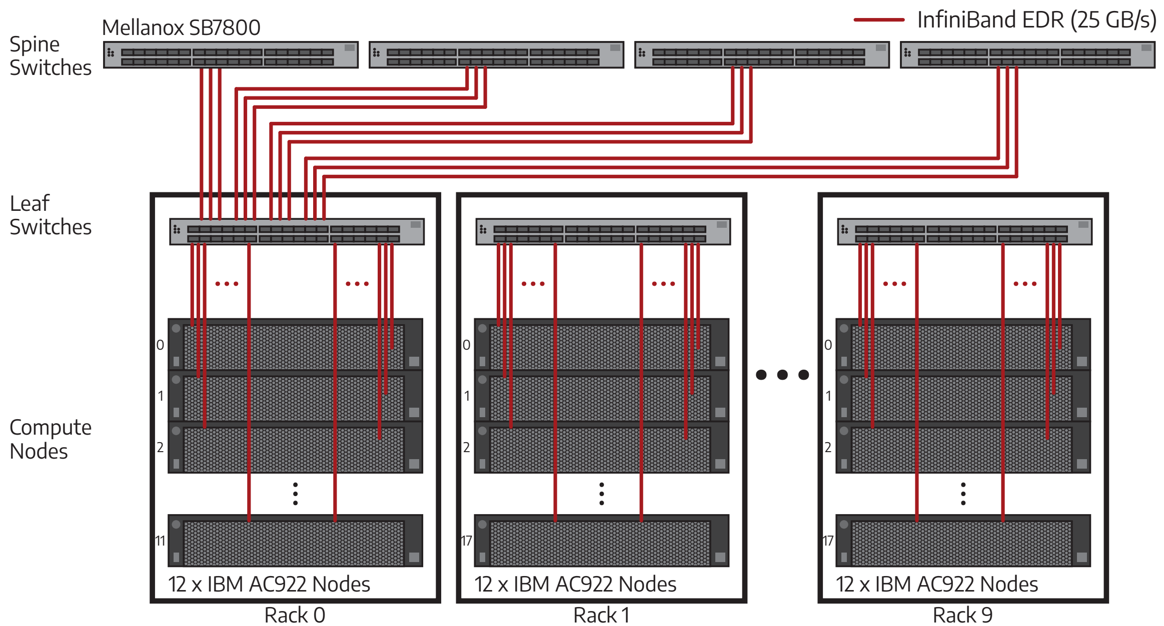


Figure 3.3: A simplified diagram of the interconnect topology on TACC Longhorn cluster, consisting of two-level leaf-and-spine switches. Each rack has a leaf switch, connecting to 12 compute nodes, each with two downlinks. Each leaf switch has twelve uplinks to connect to four spine switches, each with three uplinks. This is an approximated topology since TACC does not disclose its cluster network topology.

leaf switch is connected to four spine switches, each with three uplinks for 12 uplinks. With this configuration, communication between compute nodes within a rack has a higher bandwidth than communication between compute nodes across the rack. Although the SLURM workload manager will try to allocate the nodes on the same rack, it may not be guaranteed.

3.2.3 TACC Lonestar6 Cluster

TACC Lonestar6 comprises CPU-only nodes and GPU nodes. The GPU nodes replace the Maverick2 and Longhorn, providing access to the newer NVIDIA A100 GPUs. There are 84 GPU nodes, each equipped with two AMD EPYC 7763 CPUs, 256 GB DDR4 3200 MHz RAM, and three NVIDIA A100 40 GB PCIe GPUs. The research on this dissertation uses Lonestar6 as an early exploration of new GPU architecture, migrating the code bases to a new environment. However, experimental results are obtained from Maverick2 and Longhorn unless otherwise noted. Texas Advanced Computing Center (2024) provides the user guide for Lonestar6.

3.2.4 Other Computing Resources

With the decommissioning of Maverick2, I have to find another computing resource that provides the same NVIDIA Tesla V100 PCIe GPU for developing a communication-reducing algorithm (Chapter 6). While the Lonestar6 is available with a newer GPU, all experiments must be re-run on the new platform to get updated results for comparing the effectiveness of communication-reducing algorithms. Therefore, I built a desktop PC with two NVIDIA Tesla V100 16 GB GPUs, the same model as the one in TACC Maverick2. It has a single AMD Ryzen Threadripper 3970X CPU and 256 GB of DDR4 3600 MHz RAM. The interconnect topology is simple: one CPU is connected to two GPUs using PCIe x16 interfaces. This desktop provides a sufficient environment for developing the communication-reducing algorithm.

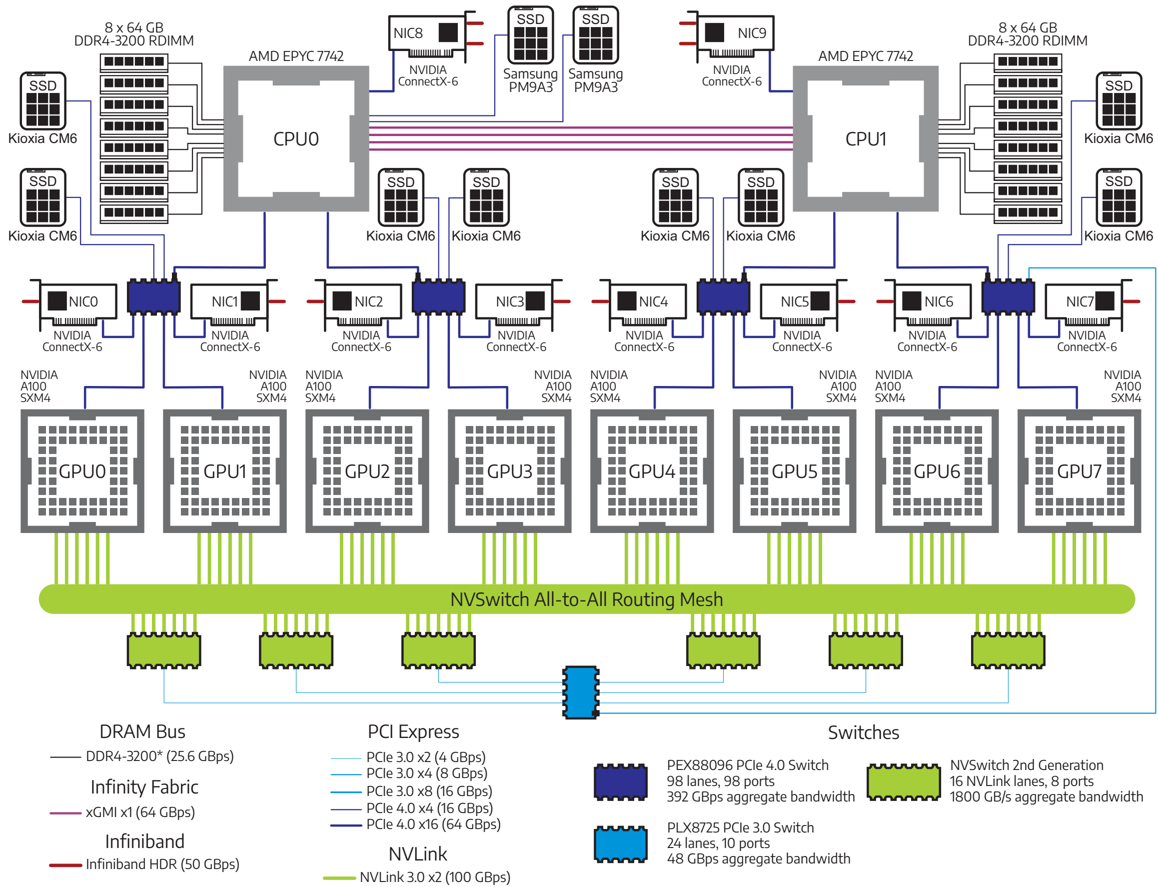


Figure 3.4: A simplified diagram of the interconnect topology inside NVIDIA DGX-A100 platform. Two AMD EPYC 7742 CPUs, each equipped with eight-channel DDR4-3200. Eight NVIDIA A100 GPUs are connected using NVLink 2.0 through NVSwitch All-to-All routing mesh.

Another platform this dissertation uses is the NVIDIA DGX-A100, provided by NVIDIA Corporation (2020b). This platform is designed to tackle demanding machine learning applications. Thus, it has various high-bandwidth interconnects to make sure every component in the platform is interconnected as fast as possible. The topology of the platform is shown in Figure 3.4. While it can be intimidating when looking into the "simplified" topology, I highlight some essential features. It has two AMD EPYC 7742 CPUs connected through four InfinityFabric inter-socket links. Eight NVIDIA A100 SXM4 40 GB GPUs are connected to each other through NVSwitch 2.0 using NVLink 3.0. Unlike the node topology of Longhorn requiring the use of inter-socket CPU link for communication between pairs of GPUs, shown in Figure 3.2, each GPU in DGX-A100 has 600 GBps bidirectional bandwidth to each other GPU.

3.3 Memristor-based Processing-in-Memory Models

Since the memristor-based PIM used in Chapter 7 of this dissertation is an emerging computing technology that is still in its infancy, no actual hardware can be used to perform the experiments. Instead, the simulation framework is built by integrating two previous works: FloatPIM by Imani et al. (2019a) and NVSim by Dong et al. (2012). The energy model is taken from earlier work by Imani et al. (2020).

3.3.1 PIM Simulation Software

The FloatPIM is used as a PIM cycle-accurate simulator to estimate the performance of the memristor-based PIM chip when running the wave simulation. It provides the number of cycles to execute the wave simulations for a given data layout and operations. However, the data needs to be laid out manually into the memristor arrays, and the execution timeline needs to be defined manually since the software does not come with a compiler to translate high-level programming language into

the executable binary for PIM. Table 3.2 shows the energy (E) and time (T) model parameters of the basic operations of PIM taken from FloatPIM.

E_{set}	E_{reset}	E_{NOR}	E_{search}	T_{NOR}	T_{search}
23.8 fJ	0.32 fJ	0.29 fJ	5.34 pJ	1.1 ns	1.5 ns

Table 3.2: The energy (E) and time (T) parameter model obtained from FloatPIM for various basic operations on PIM, which includes `set`, `reset`, `search`, and bitwise logic `nor`.

3.3.2 Non-Volatile Memory Circuit Models

NVSim is used to obtain the model parameters of a non-volatile memory circuit. These data are then used to model the latency and area of the circuits inside the memristor-based PIM used in this dissertation. The breakdown of PIM read latency obtained from NVSim is given in Table 3.3, where a bank is assumed to have 4×4 mats, a mat is considered to have 16×16 subarrays and a subarray is supposed to have 1024×1024 memory cells.

Latency	**Value**
Read	40.471 ns
H-Tree	39.047 ns
Mat	1.424 ns
Mat-Predecoder	359.727 ps
Mat-Subarray	1.064 ns
Mat-Subarray-Row Decoder	206.298 ps
Mat-Subarray-Bitline	43.540 ps
Mat-Subarray-SenseAmp	805.733 ps
Mat-Subarray-Mux	8.819 ps
Mat-Subarray-Precharge	390.522 ps

Table 3.3: The breakdown of PIM read latency obtained from NVSim. Note that the subarray size is 1024×1024, mat consists of 16×16 subarrays, and bank consists of 4×4 mat.

In NVSim, each non-volatile memory chip can have multiple banks. Bank is the top-level structure modeled by NVSim and can operate independently since it is a fully functional memory unit. Inside each bank, multiple mats are connected using H-Tree

interconnect topology or bus-like interconnect topology. Multiple mats inside a bank can operate in parallel, executing a memory operation. Each mat comprises multiple subarrays and one pre-decoder block. Subarray is the basic structure modeled in NVSim, and it contains the memory cells and the peripheral circuitry, including row decoders, column multiplexers, and output drivers.

The hierarchical terminology for the memory used in this dissertation is slightly different than the one used in NVSim. In this dissertation, the memory bank is called the memory tile, while the memory mat is called the memory block. Apart from the differences in terminology, they carry the same functions and structures.

3.3.3 Energy Models

The energy model is derived from DUAL, a previous work by Imani et al. (2020), given in Table 3.4 for a PIM chip with 2 GB. Power consumption for other circuits, such as the routing switches, central controller, and host CPU, is obtained using Synopsys PrimeTime, as demonstrated by Bhatnagar (2002).

Components	Param	Value	Power
Crossbar Array	size	1 Mb	6.14 mW
Sense Amp	number	1K	2.38 mW
Decoder	number	1	0.31 mW
Memory Block	number	1	8.83 mW
Memory Tile	num_block	256	1.57 W
H-tree Routing Switch	number	85	107.13 mW
Bus Routing Switch	number	1	17.2 mW
Memory Tile Capacity	size	32 MB	1.68 W (H-tree) 1.59W (Bus)
Central Controller	number	1	6.41 W
Host CPU	number	1	3.06 W
Total	size	2 GB	115.02 W (H-tree) 109.25 W (Bus)

Table 3.4: The energy model for PIM chip with 2 GB capacity, taken from DUAL. Power consumption for other circuits, such as the routing switches, the central controller, and the CPU host are obtained from Synopsys PrimeTime.

3.3.4 PIM Platform Configurations

Following the format of Table 3.1, Table 3.5 shows the platform configurations for the memristor-based digital processing-in-memory system. This dissertation uses four PIM chips manufactured at 28 nm process nodes and operates at 900 MHz. They have different capacities of 512 MB, 2 GB, 8 GB, and 16 GB. However, they have the same memory block size, comprising 1024×1024 memory cells. They also have the same memory tile size, comprising 256 memory blocks. The only difference is the number of memory tiles in a PIM chip, resulting in different chip capacities. They all have a single-core ARM Cortex-A72 as the host CPU and 16 GB HBM2 off-chip memory, providing 900 GBps bandwidth. The off-chip memory is assumed to consume 36.91 W of power, as shown in prior work by Li et al. (2018c) while the host CPU is assumed to consume 3.06 W, as shown in Table 3.4.

The FP32 peak throughput of the PIM chip is determined based on the maximum parallelism it can achieve, the arithmetic operation latency, and the operating frequency of the chip. For example, the maximum parallelism for a 2 GB PIM chip with 1024×1024 memory block size is $2\ GB/1,024\ b = 16\ M$. The arithmetic latency is obtained from previous works by Haj-Ali et al. (2018); Imani et al. (2019a), assuming the workloads have 50% addition and 50% multiplication operations. The operating frequency, as shown in Table 3.5, is 900 MHz.

The interface between the host CPU and PIM chip is a custom interface assumed to provide sufficient bandwidth to avoid becoming the system bottleneck. The interface is only used to send commands to the PIM chip; thus, the energy consumption and performance impact are negligible. The data sharing between the host CPU and the PIM chip uses the off-chip HBM2 memory. This dissertation only considers single compute node with only one PIM chip.

Platform	**Memristor-based PIM**			
Host CPU				
Model (# Sockets)	ARM Cortex-A72 (1)			
Base / Turbo Clock	900 MHz / 900 MHz			
Total Cores / Threads	1 / 1			
Inter-socket Link	N/A			
Accelerator (PIM)				
Model	PIM-512	PIM-2G	PIM-8G	PIM-16G
# PIMs per node	1	1	1	1
Form Factor	Custom	Custom	Custom	Custom
Process Node	28 nm	28 nm	28 nm	28 nm
Clock Frequency	900 MHz	900 MHz	900 MHz	900 MHz
Memristor Array Size	**512 MB**	**2 GB**	**8 GB**	**16 GB**
L2 Cache Size	N/A	N/A	N/A	N/A
Memory Size and Type	16 GB HBM2	16 GB HBM2	16 GB HBM2	16 GB HBM2
Memory BW	900 GBps	900 GBps	900 GBps	900 GBps
# FP32 Cores	N/A	N/A	N/A	N/A
FP32 Peak Throughput	1.7 TFLOP/S	6.6 TFLOP/.S	26.4 TFLOP/S	52.8 TFLOP/S
CPU-to-PIM Link	Custom	Custom	Custom	Custom
PIM-to-PIM Link	N/A	N/A	N/A	N/A
Computation Nodes				
# Nodes (# PIMs)	1 (1)			
Inter-node Link	N/A			

Table 3.5: The compute platform (node) configurations for Memristor-based Processing-in-Memory (PIM). Note that these platforms are only constructed in a simulator since the hardware is unavailable. Only single-node configuration is considered, eliminating the need for inter-node communication.

3.3.5 Process Node Scaling

As shown in Table 3.5, the PIM chip is modeled as it is assumed to be manufactured in a 28 nm process node. However, the GPUs on the TACC Maverick2 cluster are manufactured using 16 nm or 12 nm process nodes, as shown in Table 3.1. Comparing GPU and PIM manufactured in different process nodes for performance evaluations, discussed later at Section 7.4, is not a fair comparison. Smaller process nodes have the advantages of having higher performance and lower energy consumption. The approximate scaling suggested by Ankit et al. (2019); Zuloaga et al. (2015) is applied to the PIM system to make a more fair comparison. Scaling from 28 nm to 12 nm process node allows the PIM chip to get 3.81×performance improvements and 2.0×lower energy consumption.

3.4 Measurement Tools and Computing Libraries

This section discusses the tools for gathering performance and energy data from the target hardware. For real hardware, like CPUs and GPUs, the manufacturers provide the tools. There is no real hardware available for PIM; thus, the data comes from the models plugged into the PIM simulators, as discussed in Section 3.3. The tools covered in this section include GPU profiling tools, power measurement tools, performance measurement tools, and computing libraries.

3.4.1 Profiling GPU Kernels

NVIDIA GPU with Turing or older architecture uses NVIDIA Profiling Tool (`nvprof`) to profile the GPU kernels, as shown in the user guide by NVIDIA Corporation (2024). This includes the Volta (i.e., NVIDIA Tesla V100) and Pascal (i.e., NVIDIA Tesla P100, NVIDIA GeForce GTX 1080 Ti) architectures used in this dissertation. Below are several metrics collected using the `nvprof` as a part of this dissertation's performance evaluation of GPU kernels.

- *Instruction Count.* The instruction count per warp is obtained through metric `inst_executed`. However, this metric only gives the number of warp instructions executed. Since each warp consists of 32 threads (Section 2.6.2), the total instructions executed by all threads can be approximated by multiplying the value from `inst_executed` with 32.

- *Double Precision Floating-Point Operation Count.* The total FP64 operation count is obtained through metric `flop_count_dp`. This metric includes addition, multiplication, and fused-multiply-add operations involving double precision floating point format.

- *Single Precision Floating-Point Operation Count.* The total FP32 operation count is obtained through metric `flop_count_sp`. This metric includes addition, multiplication, and fused-multiply-add operations involving double precision floating point format. However, special functions, such as square root, trigonometric, hyperbolic, and inverse operations, are not included. These special functions' floating-point operations can be obtained through metric `flop_count_sp_special`. Adding the values from these two metrics results in the total of single precision floating point operations performed.

- *Off-Chip Memory Data Transfer.* The total data transfer for both read and write is obtained through metric `dram_read_bytes` and `dram_write_bytes`, respectively. The number represents the total bytes read from the DRAM to the L2 cache and the total bytes written from the L2 cache to the DRAM. This data calculates the FLOP/byte required when performing roofline analysis (Section 5.5.3).

- *Kernel Execution Time.* The kernel execution time is obtained from `nvprof` running in summary mode. It provides the number of kernel calls and the average, minimum, and maximum time needed to execute the kernel across all kernel calls.

Newer GPU architecture, such as Ampere (i.e., NVIDIA A100), needs to use the newer profiling tool called Nsight Compute (`ncu`), as described in the user guide by NVIDIA Corporation (2023a). Some of the metrics may change, and since the NVIDIA A100 is only used for early exploration on newer hardware, the use of `ncu` is not discussed here.

3.4.2 Measuring Power Consumption

In addition to evaluating performance, understanding the power consumption and energy usage of GPU-driven applications is crucial for comparing the energy efficiency of GPUs with other accelerators. There are generally three methods for measuring GPU power: 1) using on-chip and onboard sensors made available by the drivers, 2) utilizing data from the power supply, and 3) employing external probes. Attaching the external probes may not always be possible since physical access to the compute node of TACC clusters is prohibited. In addition, obtaining the data from the power supply through its simple management bus (SM-Bus) is not always possible since it may require root access or access to the out-of-band (IPMI) interface, which TACC does not provide. In this dissertation, the NVIDIA System Management Interface (`nvidia-smi`) is used to measure the average power and energy consumption of applications running on the GPU by accessing the onboard sensors. As shown in the documentation by NVIDIA Corporation (2020c), this tool comprehensively monitors NVIDIA GPUs, including power, temperature, frequency, and utilization.

For a more comprehensive comparison with the PIM platform, the host CPU power must be included in addition to the GPU power. Intel provides a tool called Running Average Power Limit (RAPL) for measuring the CPU power, described by David et al. (2010). RAPL is a low-level interface that measures energy consumption for a CPU (and other components) without needing extra hardware. Since 2017, it has been available on most computing devices, including non-Intel processors like AMD. Finally, the power consumption for PIM hardware is obtained from the energy model

plugged into the PIM simulation. Section 3.3.3 describes how the energy modeling on PIM hardware is done.

3.4.3 Measuring Performance

At the end of the wave simulation, the time spent in each simulation kernel is displayed as a part of diagnostic and reporting (Appendix B.2.2). The time is measured using the wall clock provided by the MPI library (i.e., `MPI_Wtime`). The time data is used to measure and compare the simulation performance of both CPU and GPU codes. For roofline analysis (Section 2.5.2), the kernel execution time obtained from `nvprof` is used to measure the kernel performance instead (Section 3.4.1) since more detailed measurement at kernel granularity is needed. This data is also used to compare the performance of the GPU kernel with the PIM implementation. Finally, the kernels' performance on PIM is measured by the number of cycles given by the cycle-accurate PIM simulator.

3.4.4 Computing Libraries

The experiment uses the available computing libraries provided at the TACC clusters. This includes the CUDA Toolkit 10.2 for Maverick2 and Longhorn clusters. On the Longhorn, `gcc` 8.4.0 is built from source since the available `gcc` is too old. The `Cmake` is also built from source, aiming for version 3.18, which supports the CUDA code base. Several MPI libraries are available on both TACC clusters. Intel MPI and MVAPICH2-GDR are available on Maverick2, while SpectrumMPI and MVAPICH2-GDR are available on Longhorn. Meanwhile, the OpenMPI is unavailable and needs to be built from the source with CUDA support. The OpenMPI version 4.1.1 is built with UCX version 1.11.2 and `gdrcopy` version 2.3.

Chapter 4: Large-scale dG-based Wave Simulations on CPUs

In this chapter[1], I describe the CPU code of the dG-based wave simulations, which becomes the baseline of this dissertation. Some diagrams and explanations are adopted with modifications from my Master's Thesis, Hanindhito (2020), an early study of this dissertation.

First, I briefly explain how the simulation mesh is generated using the `p4est` by Burstedde et al. (2011), a parallel adaptive mesh refinement library used by the CPU code (Section 4.1). I explain some functions provided by `p4est` to manipulate the mesh and iterate through elements, faces of elements, and corners of elements. Although I often call these functions using the prefix `p4est` used for the 2D spaces, these functions can be called with the prefix `p8est` for 3D spaces, which are the focus of this dissertation. Secondly, I describe the data structures that are important to be explained (Section 4.2), as they play crucial roles in developing the GPU code in Chapter 5. These include the data structures defined by the `p4est` and the custom data structure defined by the wave simulation application.

Third, I explain the simulation flow and the data flow of the wave simulations, which are generic to both acoustic and elastic wave simulations (Section 4.3). Next, I describe the simulation kernels used in this simulation, including the mass inverse, volume, flux, and integration kernels (Section 4.4). I also discuss the multi-CPU implementation leveraging `p4est` capability to partition and balance mesh across multi-processor systems and clusters of compute nodes, adding scalability to the sim-

[1] Some materials on this chapter are adapted with modification from my Master's Thesis: Bagus Hanindhito. GPU-accelerated high-performance computing for architecture-aware wave simulation based on discontinuous Galerkin algorithms. UT Electronic Theses and Dissertations, 2020. `http://dx.doi.org/10.26153/tsw/42690`.
My Master's Thesis only considers acoustic wave simulations; thus, the modifications are performed to make the explanation more general and applicable to elastic wave simulations.

ulation to handle industry-relevant larger problem sizes (Section 4.5). Finally, at the end of this chapter, I briefly describe the method to verify the numerical accuracy of the simulation (Section 4.6).

For interested readers who want more details on the CPU code, I have prepared a more detailed explanation of the CPU code in Appendix B. The appendix further explains the data structure, the simulation application and kernels' high-level flow diagram, and the simulation configuration options.

4.1 Mesh Generation and Partitioning

Before running the wave simulation, the problem domain (e.g., ocean) is discretized into several elements as shown in Figure 4.1. The elements are called quadrants or octants for 2D or 3D problems, respectively. The number of elements depends on the refinement level that is used; the higher the level, the finer the mesh is (Section 4.1.1). Each element comprises several nodes in which the unknown values in the wave equations (Section 2.2) are evaluated. The `p4est` generates, handles, partitions, and distributes the mesh. Note that, in this work, the mesh is generated once at the beginning, and its structure remains constant throughout the simulation run.

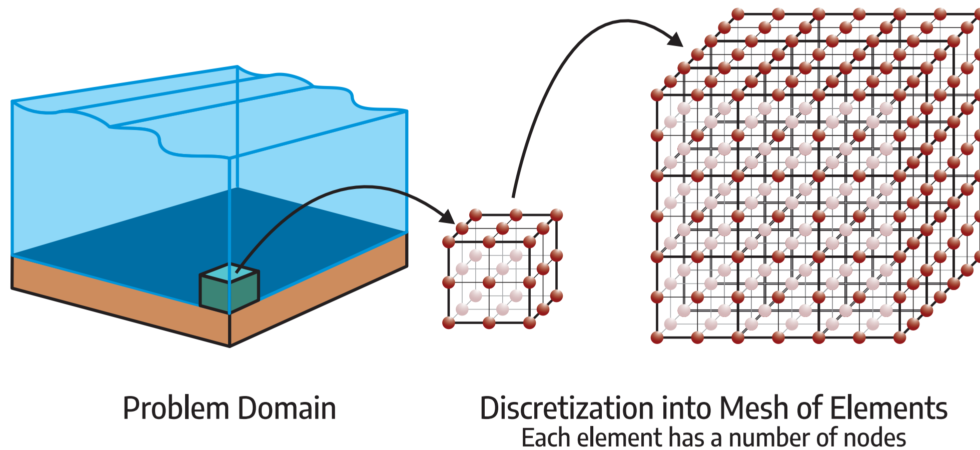


Figure 4.1: The problem domain (e.g., ocean) is discretized into several elements, resulting in a mesh of elements. Each element is comprised of several nodes.

The discretization itself results in inherent parallelism of the applications, which can be harnessed on particular hardware to improve simulation performance (Section 4.1.4). Larger problem domain consisting of vast amount of elements may not fit inside single compute node, and thus the mesh of elements is partitioned and distributed across multiple compute nodes to run large-scale simulation (Section 4.1.5).

4.1.1 Refinement of Meshes

The refinement level is one of the runtime configurations (Appendix B.6), and it controls the refinement of the mesh used for simulation. A higher refinement level produces finer mesh consisting of a higher number of smaller elements. This can improve the numerical fidelity of simulation and be useful for evaluating high-frequency waves but with higher computation costs (Appendix A.3.3).

Figure 4.2 illustrates the effect of using higher refinement level in 3D space. The 0th refinement level means only one element represents the problem space. The element has a width, height, and length of 1. Increasing the refinement level to 1 divides this large element into eight smaller elements, each with width, height, and length of 0.5. These elements are called level-1 elements, and the numbering of the elements follows the `p4est` numbering scheme, explained later in Section 4.1.3. Moving to a higher refinement level of 2 further divides the level-1 elements into eight smaller elements, each with a width, height, and length of 0.25. In uniform mesh (Figure 2.3), where the size of elements is the same across the mesh, there are 64 level-2 elements at this refinement level. Likewise, a refinement level of 3 divides all 64 level-2 elements into 512 level-3 elements.

Note that the refinement level affects the value of the time step to run the simulation. A higher refinement level requires a smaller time step to maintain the numerical stability. This means that with a higher refinement level, the simulation needs to run for more time steps to reach the same simulation end time, a trade-off between having higher numerical fidelity and the total run time required.

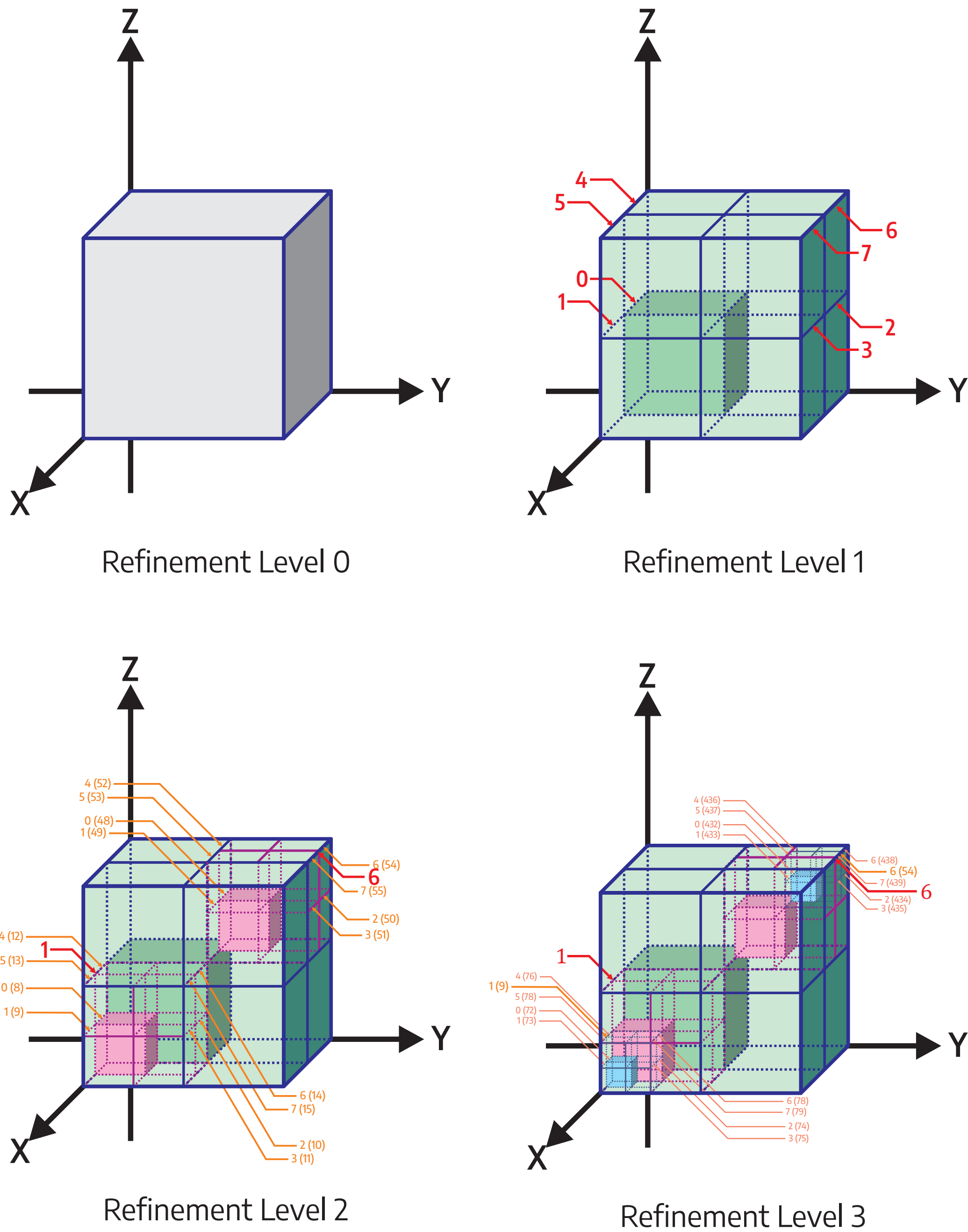


Figure 4.2: The refinement level is a runtime configuration that controls the refinement of the computational mesh. A higher refinement level discretizes the problem domain into a larger number of smaller elements.

The mesh can have a non-uniform refinement level, resulting in non-conforming meshes (Figure 2.3). This allows a higher refinement level to be used on the part of the mesh where higher fidelity is required to achieve the required accuracy. For example, part of the mesh where two or more different materials met (i.e., boundary) or where the rate of change of variable is significant. The `p4est` used in this work can manage non-conforming mesh. However, since this dissertation only focuses on uniform mesh, the refinement level is uniform across the mesh.

4.1.2 Elements of Mesh

The previous section, Section 4.1.1, has discussed how the problem domain is discretized into many smaller elements. A higher refinement level brings more numerical fidelity and accuracy to the simulation since each element covers smaller problem spaces. Thus, it can better handle abrupt changes in unknown values, resulting in more accurate solutions. However, in large problem sizes, the space covered by an element may still be too large if the unknown values (Section 2.2) are evaluated at one point per element. Therefore, each element has multiple nodes, where the unknown values are evaluated: four for acoustic wave equations and nine for elastic wave equations in 3D space.

Figure 4.3 illustrates the elements in both 2D and 3D spaces with their nodes. Although, in the illustration, the nodes are evenly spaced, mathematically, they are not. The Gauss-Lobatto-Legendre (GLL) Integration Scheme used in this work (Section 2.3.2) dictates the location of these nodes, making them more spread closer to the center of the element and more clustered closer to the side of the element. An element can have many nodes, called higher-order elements (Appendix A.3.3). Higher-order elements will have better arithmetic intensity since the ratio between interior nodes (i.e., nodes not located at the face of elements) and face nodes is higher, meaning more local computation (i.e., volume computation, Section 4.4.2) compared to external computation (i.e., flux computation, Section 4.4.3), which will be discussed in Section 5.5.4. However, having a higher-order element reduces the distance between

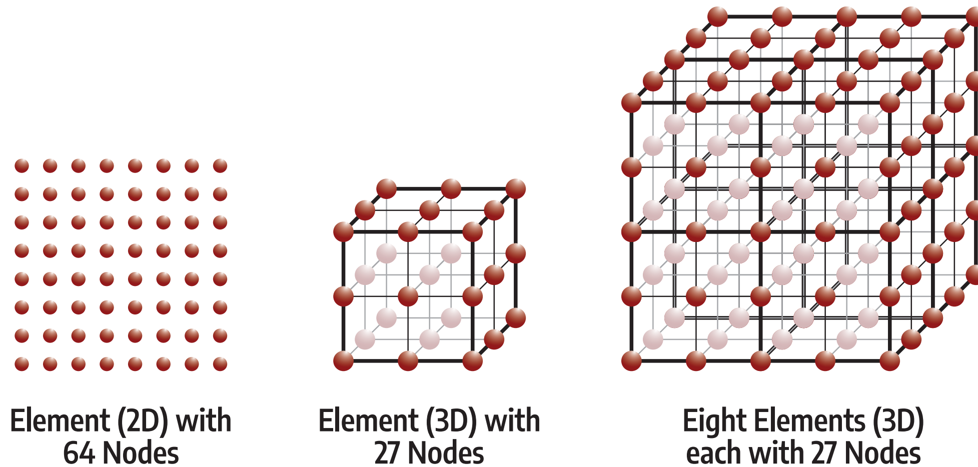


Figure 4.3: The arrangement of nodes (i.e., evaluation nodes) of elements, either in 2D (quadrant) or 3D (octant). Note that while the illustration shows the nodes are evenly spaced, mathematically, they are not, which is attributed to the Gauss-Lobatto-Legendre (GLL) Integration Scheme used in this work (Section 2.3.2.3).

the nodes, requiring smaller time steps to preserve numerical stability. This means the simulation will need to run for more time steps to reach the same simulation end time. It is a trade-off between having a higher numerical fidelity at the expense of a longer time to complete the simulation.

Throughout this work, the number of nodes in the element is called `NNODE`, where `NNODE`=`NODE_1D`^{3} for 3D space. The number of nodes located in every face of the element is called `NNODE_FACE`, where `NNODE_FACE`=`NODE_1D`^{2} for 3D space. The number of nodes in each element is configurable at compile-time (Appendix B.5). I choose `NODE_1D`=8 to make it more hardware-friendly, resulting in 512-node elements, since power-of-two makes managing the memory access easier and more cache-friendly while having the numerical fidelity needed. In addition, it also makes it easier to distribute the work across threads in GPU (Chapter 5) since the number of threads in a wavefront (AMD) or a warp (NVIDIA) is multiple of 64 or 32, respectively. If the number of nodes in an element is not divisible by 64 or 32 (i.e., the remainder is not zero), several dummy threads will be on a wavefront/warp that does nothing.

4.1.3 Element Numbering Schemes

The numbering of elements, faces, and nodes follows the numbering scheme used by `p4est`. It starts by traversing in the direction of the x-axis and then continues to the y-axis and the z-axis. For element, the numbering scheme is shown in Figure 4.2. For example, at 2nd refinement level, the 0th starts at the element at coordinate (x, y, z) of $(0, 0, 0)$. Then, moving to the positive direction along x-axis is the 1st element $(0.5, 0, 0)$. Since no more elements are along the x-axis at $(y, z) = (0, 0)$, the y-axis is advanced by 0.5 to the positive side. Now, the 3rd and 4th elements can be labeled for elements at $(0, 0.5, 0)$ and $(0.5, 0.5, 0)$, respectively. Since no more elements are along the xy-plane at $z = 0$, the z-axis is advanced by 0.5 to the positive side. The strategy is then applied to 5th $(0, 0, 0.5)$, 6th $(0.5, 0, 0.5)$, 7th $(0, 0.5, 0.5)$, and 8th $(0.5, 0.5, 0.5)$ elements.

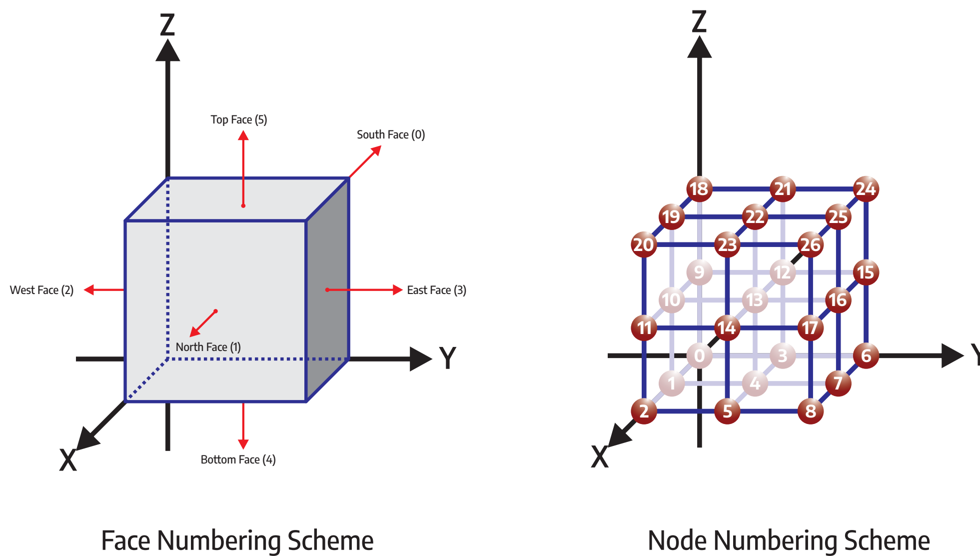


Figure 4.4: The numbering scheme for faces and nodes of an element, following the numbering scheme of `p4est`.

Figure 4.4 shows the numbering scheme for an element's faces and nodes. As explained in the previous paragraph, the strategy is the same as numbering the elements. Following this convention is crucial, especially for determining the neighboring

elements during flux computation (Algorithm 3, Figure 5.1), determining the faces that touch each other, and determining the location of the node on the face of the element.

4.1.4 Source of Parallelism

As a result of discretization, two primary sources of parallelism can be extracted to improve the performance of the wave simulations, as shown below.

- *Element-level Parallelism.* Due to dG discretization (Section 2.3.1), during the local computation (i.e., compute volume, Section 4.4.2), the elements can be computed individually, and thus the computation can be performed in parallel.
- *Node-level Parallelism.* The evaluation of unknown values in each node within an element can be done in parallel.

4.1.5 Partitioning and Distributing Mesh

In large problem sizes with vast amounts of elements, single compute node may not be sufficient to hold all of the elements. Therefore, the mesh is partitioned and distributed across multiple compute nodes, making the simulation scalable. Partitioning the mesh also allows a subset of mesh to run on multiple CPU cores, improving simulation performance through parallel execution. The `p4est` library will partition and distribute in a balanced manner, meaning that each CPU core handles roughly the same number of elements.

As shown in Figure 4.5, in a multi-processors execution, `p4est` partitions the mesh into several sub-meshes, each handled by one MPI process that runs on one CPU core. The color shows the `MPIRank` where these elements live. If the elements need neighboring elements' data that lives in another MPI process, the communication between the MPI process needs to be made. It is called ghost exchange, and additional data structures (i.e., ghost buffer and ghost layer) are needed (Section 4.5). These

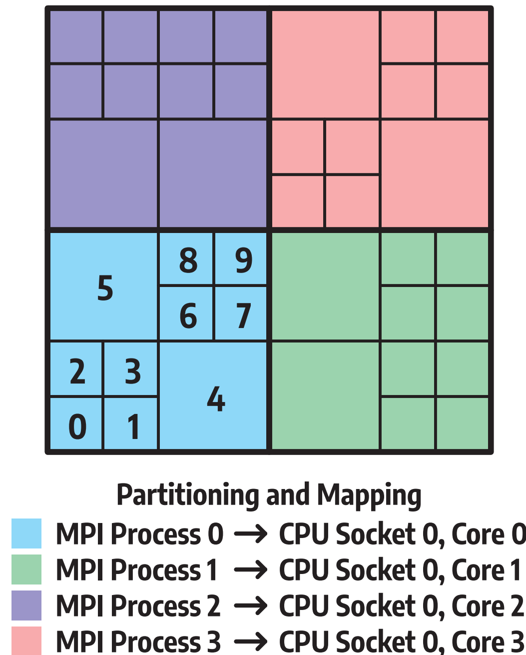


Figure 4.5: The `p4est` partitions the problem domain into multiple elements called quadrants (2D space) or octants (3D space). Although the illustration uses non-conforming mesh, it also applies to the uniform meshes, the only mesh considered in this dissertation. In multi-processor systems, `p4est` partitions and distributes the elements into each MPI process running on a dedicated processor (core) in a balanced manner (i.e., each MPI process handles roughly the same number of elements.

features are provided by `p4est`, allowing the wave simulation to scale to multi-socket systems or even multi-node clusters to handle larger problem sizes.

4.2 Data Structure

While the wave simulation application uses many data structures, three are the most important to discuss: two from the mesh handling library `p4est` and one from the custom user data representing each element. Figure 4.6 shows the high-level view of these two data structures used by `p4est`: `p4est_tree_t` to store mesh data and `p4est_mesh_t` to store mesh structure. The custom user data, `ElementDataBase`, represents each element and is attached into `p4est_quadrant_data`. For functions

and data structures used by p4est, it starts with prefix p4est and p8est for 2D and 3D space, respectively. Although the 2D space is used to simplify explanation, the 3D space follows the same principle. Additional data structures, such as the ghost buffer and the ghost layer, will be explained in Section 4.5.

4.2.1 Mesh Data

The p4est_tree stores the data for each element using an array of quadrants[2], p4est_quadrant_t, as shown in Figure 4.6. The array itself is implemented using sc_array_t, which is part of the libsc library used by p4est. Inside each element of the array, there is a void pointer to user_data, where the user can provide their custom data structure that represents each element. In this dissertation, each element is represented using a struct called ElementDataBase, which will be explained in Section 4.2.3.

4.2.2 Mesh Structure

The p4est_mesh_t stores the structural information of the mesh (e.g., how each element faces the other). A mesh can have multiple trees, which is called a forest. As its name suggests, the structure of each tree is represented as a tree data structure implemented using linked lists. In simulation runs using multiple compute nodes and CPU cores, the p4est will partition the mesh into sub-meshes, each containing roughly the same number of elements. In the case of Figure 4.6, the mesh is partitioned into four sub-meshes, following the partitioning strategy given in Figure 4.5.

Iterate over the (sub)-mesh structure is done using the provided function, p4est_iterate. This function will automatically execute a user-supplied callback function, which is a function that needs to be performed for each object inside the (sub)-mesh. Note that each MPI process executes this function and iterates only over

[2]quadrant in 2D space, octant in 3D space.

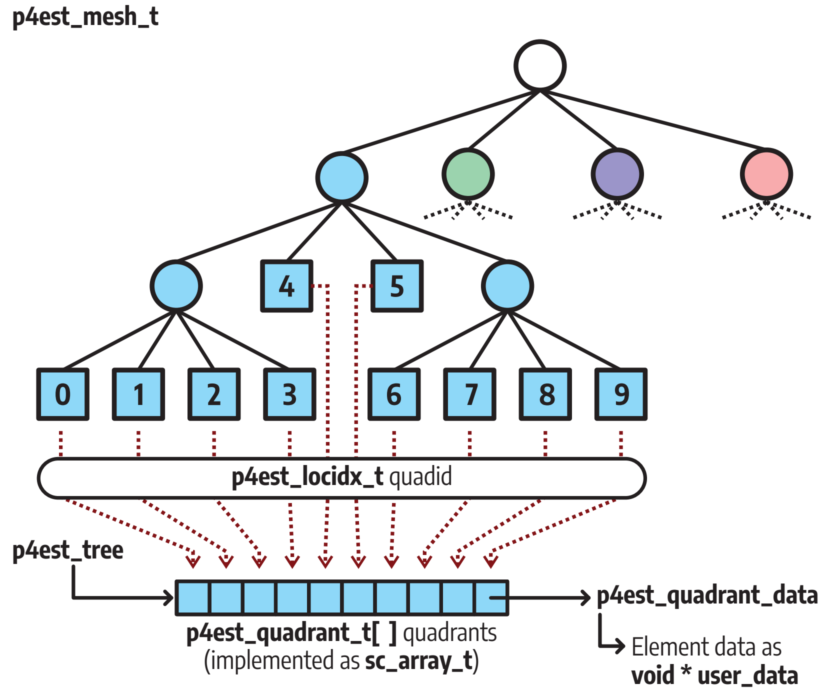


Figure 4.6: To keep track of each element, `p4est` uses two primary data structures: `p4est_mesh_t` to store the mesh structure representing how each element interacts with others, and `p4est_tree` to store the data of each element. Many operations, including volume and flux computation, can be performed on each element by traversing the mesh structure using pointers, such as through the provided function `p4est_iterate`.

the sub-mesh given to that process. Three different iteration objects alongside their callback function are supported by `p4est_iterate`, which are explained below.

- `iter_volume`: Iterate through every quadrant/octant interior and execute the callback function given as `p4est_iter_volume_t`. In this case, `p4est` will execute the callback function, such as the kernel to perform volume computation (Section 4.4.2), for every element, serially. This is illustrated in Figure 4.7.

- `iter_face`: Iterate through every pair of faces between neighboring quadrant/octant and execute the callback function given as `p4est_iter_face_t`. In this case, `p4est` will execute the callback function, such as the kernel to perform

flux computation (Section 4.4.3), for every pair of faces of neighboring elements, serially. This is illustrated in Figure 4.8.

- `iter_corner`: Iterate through every pair of corners between neighboring quadrant/octant and execute the callback function given as `p4est_iter_corner_t`. This iteration is not used in this dissertation.

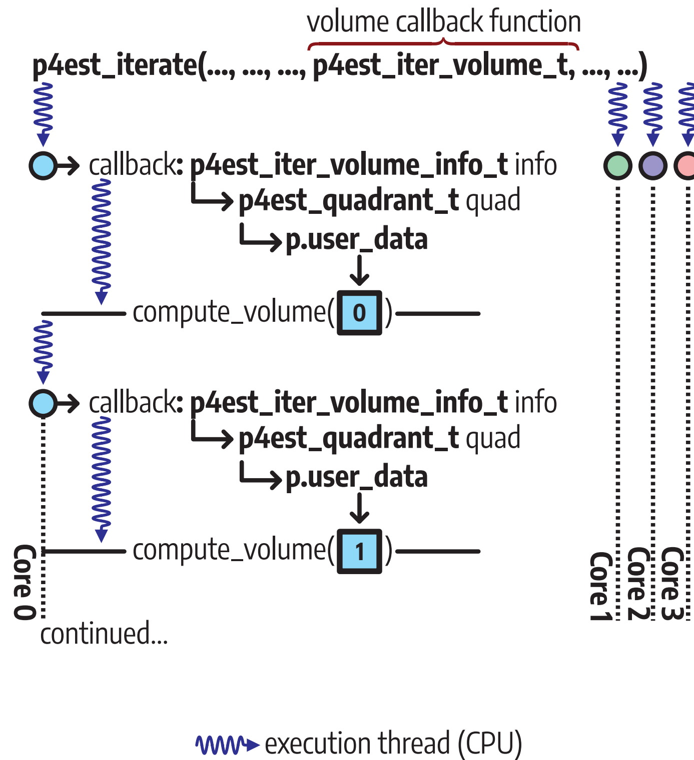


Figure 4.7: In `p4est`, iterating each element to perform volume computation is done by calling the provided function `p4est_iterate`. This function will execute the callback function given as `p4est_iter_volume_t` argument for each element one by one. Then, the associated element data can be obtained through pointer operations.

While it is possible to call `p4est_iterate` once for all three types iterations (i.e., passing `p4est_iter_volume_t`, `p4est_iter_face_t`, and `p4est_iter_corner_t` together), the work in this dissertation have separate call to `p4est_iterate` for `iter_volume` and `iter_face`. The latter requires ghost exchange to finish, and

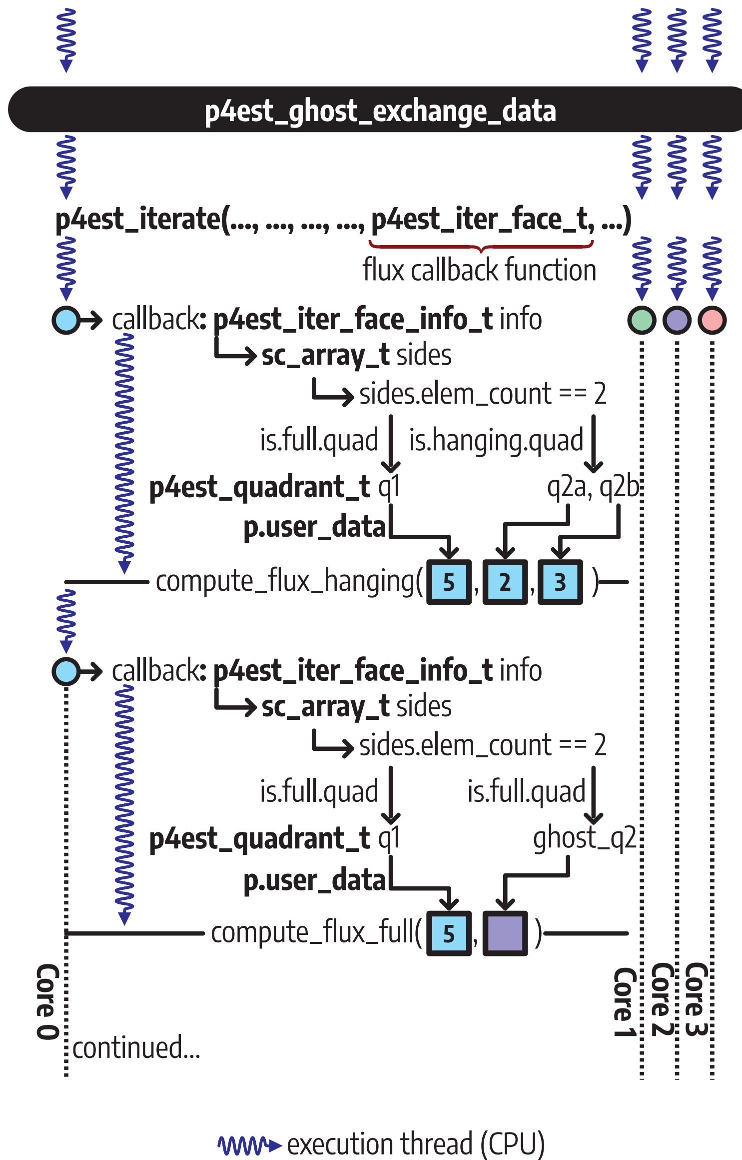


Figure 4.8: In `p4est`, iterating each group of elements that faces each other to perform flux computation is done by calling the provided function `p4est_iterate`. This function will execute the callback function given as `p4est_iter_face_t` argument for each group of elements one by one. Then, it returns the information of neighboring elements, whether the neighbor is a full element (1:1), a hanging element (1:2), or none (boundary). It also provides a pointer to appropriate storage, whether it is located in local memory (`p4est_quadrant_t`) or not (ghost element on ghost layer). Then, pointer operations can obtain the face information for each element.

thus, having separate calls for `iter_face` allows to overlap ghost exchange with `iter_volume`, as shown later in Section 4.3.1.

4.2.3 Element Database

The `user_data` attached to the `p4est_quadrant_data` shown in Figure 4.6 is a user-created data structure representing each mesh element. This user-created data structure in this dissertation is called `ElementDataBase`. While only a brief explanation is given in this section, detailed information about `ElementDataBase` can be found in Appendix B.1. In summary, `ElementDataBase` is a structure of arrays.

Among other members, two arrays of real values inside `ElementDataBase` are important to discuss: `variables` and `contributions`. The length of these two arrays depends on the number of nodes in each element (`NNODE`, Section 4.1.2) and the wave equations that are being solved (i.e., acoustic or elastic). The `variables` stores the unknown values evaluated in the simulation for each node within an element, which consist of four and nine values per node for acoustic and elastic wave equations in 3D space, respectively (Section 2.2). The `contributions` collect the result of volume computation and flux computation, which then are integrated using the time integrator (Section 4.4.4) to update the `variables` for the next time-step.

The time integrator uses `auxiliary` array inside `ElementDataBase` to store temporary values, whose length depends on the type of time integrator being used, the number of nodes in each element, and the wave equations that are being solved. The fourth-order Runge-Kutta (LSRK4) is used in this dissertation, which requires only one auxiliary variable per variable per node. The material properties are stored in `materials` array, whose length depends on the `NUM_MATERIALS` and `NNODE_MATERIALS`. As discussed in Section 2.2, `NUM_MATERIALS=2` and `NUM_MATERIALS=3` for acoustic[3]

[3] κ and ρ

and elastic[4] wave simulations, respectively. Finally, the `mass_inverse` stores the inverse of the diagonal mass matrix (Equation (A.32)), which is computed once during initialization of simulation using the mass inverse kernel (Section 4.4.1).

Struct Member	Length (# items)		Data Size (bytes)			
			Acoustic		Elastic	
	Acoustic	Elastic	FP32	FP64	FP32	FP64
`mass_inverse`	512	512	2,048	4,096	2,048	4,096
`variables`	2,048	4,608	8,192	16,384	18,432	36,864
`contributions`	2,048	4,608	8,192	16,384	18,432	36,864
`auxiliary`	2,048	4,608	8,192	16,384	18,432	36,864
`materials`	2	3	8	16	12	24
Other Members (See Appendix B.1)			6,197	12,369	6,197	12,369
Memory Alignment Padding			3	7	3	7
Total Size of Element			32,832	65,640	63,556	127,088

Table 4.1: The size of each element, represented using `ElementDataBase`, for 3D space with `NNODE_1D=8`, `NNODE=512`, `NNODE_MATERIAL_1D=1`, and fourth-order Runge-Kutta (LSRK4) time integration (i.e., `NUM_AUX=1`). Note that the total struct size includes struct padding for memory alignment.

The size of `ElementDataBase` depends on many factors that are given as compile-time configurations (Appendix B.5). In this dissertation, the simulation are compiled with `DIMENSION=3`, `NNODE=512`, `PROBLEM_TYPE`, which can be acoustic or elastic, and `NNODE_MATERIALS=1`. The element size in this dissertation is given in Table 4.1. Note that some padding is added to the struct of `ElementDataBase` for memory alignment.

4.3 Simulation Flow and Data Flow

In this section, the simulation flow and the data flow are discussed. Both are important to develop the GPU version of the simulation (Chapter 5) and the PIM version of the simulation (Chapter 7) as both versions follow closely the flow of the CPU codes.

[4] λ, μ, and ρ

4.3.1 Simulation Flow

The simulation consists of two main loops, as shown in Figure 4.9. The outer loop, denoted by the dark-green line, the light-green line, and the light-blue line, performs time-stepping; hence, it is called a time-step loop. The light-blue line executes the time-stepping and incrementing the time with the time step $\delta\ t$ until the simulation reaches the end time, as defined in runtime configuration (Appendix B.6). If the time has not reached the end time, the time-stepping proceeds with initializing the time-integrator before entering the integration loop, as denoted by the dark green line. Otherwise, the simulation concludes, and the report and diagnostic are produced, as denoted by the light-green line.

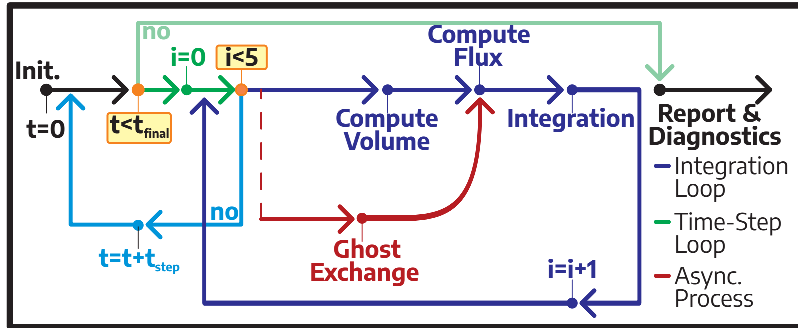


Figure 4.9: Simplified high-level simulation flow, showing two main loops: the outer time-step loop and the inner integration loop. The integration loop is executed five times per time step, launches the simulation kernels, and performs the ghost exchanges.

The inner loop, denoted by the dark-blue line, is the time integration loop, which performs the fourth-order, low-storage Runge-Kutta (LSRK4) integration (Section 2.3.3). The integration loop is repeated five times per time step. Within this loop, the simulation kernels are called, which include volume kernel (Section 4.4.2), flux kernel (Section 4.4.3), and integration kernel (Section 4.4.4). Therefore, these kernels are called five times per time step. In addition, asynchronous ghost exchange is performed for multi-CPU and multi-node runs, which will be explained in Section 4.5.

Before entering the loops, initialization is performed. The initialization includes generating and partitioning the mesh, allocating the ghost layers and ghost buffers for ghost exchange, and executing the mass inverse kernel (Section 5.3.1) to calculate the inverse of the diagonal mass matrix. At the end of the simulation, reports and diagnostics are performed, which show the numerical error compared to the analytical solution, the range of the computed solutions, and the simulation performance. Not shown in the diagram, the simulation also supports checkpointing and dumping the data for predefined time step frequency. Detailed explanation on simulation flow can be obtained in Appendix B.3.

4.3.2 Data Flow

Having discussed the `ElementDataBase` as the element's data structure in Section 4.2.3, this section connects the dots on how each member inside the data structure relates to each other. The element's data flow is shown in Figure 4.10, which applies to all elements within the mesh. The *compute volume*, *compute flux*, and *integrator* are the simulation kernels discussed later in Section 4.4.

The `variables` store the unknown values, as described in Figure 2.1. At the beginning of the simulation, initialization is made to `variables` as an initial condition. The *compute volume* and *compute flux* use `variables`, `materials`, and other members of `ElementDataBase` to compute volume contributions and flux contributions, respectively, which are accumulated in `contributions`. The *compute flux* needs the data of neighboring elements to perform flux computation while the *compute volume* is entirely local operations to the element. Then, the *Integrator* updates the `variables` for the next time step using the `contributions`, `mass_inverse` (not shown), and the `auxiliary`, which is used to store temporary values for temporal integration using the LSRK4 integrator (Section 2.3.3).

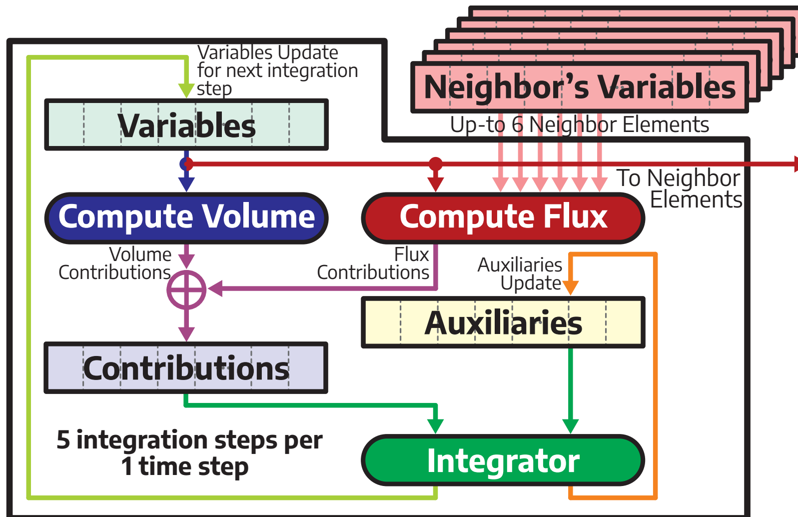


Figure 4.10: Simplified high-level data flow diagram for each element during the simulations, showing the relation between the members of `ElementDataBase` and the simulation kernels. The volume and flux kernels compute the `contributions` based on present `variables` and the `materials`, among other things. Then, the integration kernel updates the `variables` for the next time step by operating on `contributions`, `mass_inverse`, and the temporary variables stored in `auxiliary`.

4.4 Simulation Kernels

This section briefly discusses the simulation kernels used for wave simulations. Although other kernels are exactly the same, the acoustic and elastic wave simulations have different volume and flux kernels. However, they have similar execution flows, which will be explained more generically. Interested reader can find more details on each simulation kernel from Appendix B.4.

4.4.1 Mass-Inverse Kernel

The mass inverse kernel initializes the `mass_inverse` inside each element by computing the inverse of the diagonal mass matrix. The `mass_inverse` is used by the integration kernel to update the `variables`. This kernel is straightforward and is executed for each element in the mesh by giving it to `p4est_iterate` as a `p4est_iter_volume_t` callback. This kernel is only launched once during the initial-

ization of the simulation (i.e., before entering the simulation loops). Therefore, this kernel only has a very minor impact on the overall performance of the simulation. A detailed explanation of mass inverse kernel is available in Appendix B.4.1.

4.4.2 Volume Kernel

The volume kernel performs the spatial derivatives of each variable for each node within an element. By using straight-faced hexahedral elements along with the Gauss-Lobatto-Legendre (GLL) integration scheme (Section 2.3.2), the derivative calculation in each direction is decoupled from other directions, simplifying the dot-products between a subset of `variables` and constant differential vector. The derivative result is used to compute the volume contributions, which are accumulated in `contribution`

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

1 **for** *all* N^3 *nodes (i,j,k) within an element* **do**
2   **for** *offset* $o = 0, \ldots, N-1$ **do**
3   **for** *variable* $r = 0, \ldots, R$ **do** *some of*
4             $\frac{\partial u^e_{r,i,j,k}}{\partial x} \mathrel{+}= u^e_{r,(i+o)\%N,j,k} * const_dx_o$
5             $\frac{\partial u^e_{r,i,j,k}}{\partial y} \mathrel{+}= u^e_{r,i,(j+o)\%N,k} * const_dy_o$
6             $\frac{\partial u^e_{r,i,j,k}}{\partial z} \mathrel{+}= u^e_{r,i,j,(k+o)\%N} * const_dz_o$
7   **end**
8   **end**
9     $\forall r : c^e_{vol,r,i,j,k} = f(\nabla u^e_{0,i,j,k}, \ldots, \nabla u^e_{R,i,j,k})$
10 **end**

Volume kernel is the most compute-intensive kernel in the wave simulation. Fortunately, this kernel is highly parallelizable since volume computation in each element node can be performed in parallel. In addition, all operations in this kernel are local to each element, which opens another opportunity to extract parallelism. In

CPU code with p4est, the volume kernel is given as p4est_iter_volume_t callback. Although acoustic and elastic wave simulations have different volume kernels, they have generic execution flow, illustrated in Algorithm 1. Detailed explanation on volume kernel is available on Appendix B.4.2.

4.4.3 Flux Kernel

Although it is less compute-intensive than the volume kernel, the flux kernel is the kernel with the most complex execution flow. It performs the computation on the face of the element and requires the neighboring element's face data; hence, it involves non-local operations. Ghost exchange must be completed before launching this kernel in multi-processor runs, as shown in Figure 4.9.

In 3D space, an element can have up to six neighbors, one for each face; thus, the kernel is executed up to six times per element. The p4est_iterate makes it easy to iterate through each pair of faces instead of iterating through each element, by giving the flux kernel as p4est_iter_face_t callback. This automatically removes the double computation problem of flux contribution, which is present when iterating the element individually. Elements located within the boundary of the problem domain may not have neighbors for some of their faces. This is a special case that must be handled separately, which takes the boundary condition into account. At the end of the kernel execution, the flux contributions for each node located on the face of an element are accumulated into contributions.

The complex execution flow of this kernel is due to many branch operations, which are required to determine the neighboring element, determine whether the neighboring element is a ghost element, handle special cases whenever there is no neighboring element, and determine the nodes that are located on the face to access its data based on node numbering scheme (Section 4.1.3). Non-conforming mesh with 1:2 hanging elements, which are not discussed here, will have a more complex execution flow for flux, since projection and filter must be computed from the smaller

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 $\mathbf{c}_{\mathrm{f}}^{e(f1)}$, $\mathbf{c}_{\mathrm{f}}^{e(f2)'}$

```
1 for all N^2 face nodes (i,j) do
2     for variable r = 0, ..., R do
3         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, ..., R)
4     end
5 end
```

element to the larger element.

The flux computation can use different flux solvers. In this dissertation, only the Riemann flux solver is implemented for acoustic wave simulation. On the other hand, the elastic wave simulation has two flux solvers that can be chosen from: the Riemann flux solver and the Central flux solver. Although the flux kernel is problem-specific, solver-specific, and boundary-condition-specific, they have the same execution flow, illustrated in Algorithm 2. A detailed explanation of flux kernel can be found in Appendix B.4.3.

4.4.4 Integration Kernel

The integration kernel computes the time integration to update the `variables` for the next time step, using data from `contributions`, `mass_inverse`, and `auxiliary`. The integration kernel is the smallest kernel, where, judging by the amount of data it needs, memory access dominates its behavior. The kernel operates on each element and is given as `p4est_iter_volume_t` callback to the `p4est_iterate` function. Although this kernel is frequently launched (i.e., five times per time step), there is no room for optimization since the kernel is very short. Acoustic and elastic wave simulations share the same integration kernel, and only the fourth-order low-storage Runge-Kutta (LSRK4) integration (Section 2.3.3) is used in this dissertation. Further explanation on the integration kernel is available in Appendix B.4.4.

4.5 Multi-CPU Implementation

To handle industry-relevant realistic problem sizes, the wave simulation must be scalable by utilizing multiple processor cores and, eventually, multiple compute nodes. This multi-CPU and multi-node support provides more significant aggregate memory to handle larger problem sizes and improve simulation performance through parallel execution, aggregating many CPU cores to compute the solution. However, implementing support for large-scale simulation from scratch is an uphill task.

Fortunately, the `p4est`, which is used for the mesh library in this dissertation, already has features to partition and distribute the mesh to multiple processor cores, as discussed earlier in Figure 4.5. Underneath, `p4est` relies on Message Passing Interface (MPI) communication to exchange the data between many MPI processes, each running on a CPU core. The `p4est` tries to create a balanced mesh partition to each CPU core (i.e., each CPU core holds roughly the same number of elements) and provides communication functions to perform data exchanges. These data exchanges are called ghost exchanges and are performed before the flux kernel is executed.

In this section, the multi-CPU implementation is discussed. Since the implementation relies on the functionality provided by `p4est`, I will briefly touch on some of the `p4est` functions related to the ghost exchange. This includes the data structure used to perform the ghost exchange and the functions to perform asynchronous ghost exchange to hide the communication overhead by overlapping communication with computation. Familiarity with these helps develop multi-GPU support, which will be discussed in Section 5.6.

4.5.1 Data Structure

To perform the ghost exchange, `p4est` maintains two data structures: ghost layer and ghost buffer. In simple terms, the ghost layer stores the ghost elements' data received from other MPI processes. In contrast, the ghost buffer is used to stage the elements' data that become ghosts for other MPI processes before sending them.

4.5.1.1 Ghost Layer

As described earlier, the ghost layer stores the data of the ghost elements adjacent to the outermost elements on local (sub)-meshes. During the ghost exchange, the MPI receive functions (i.e., `MPI_Recv` for blocking or `MPI_Irecv` for non-blocking) use the ghost layer as receive buffer. A simplified illustration of the ghost layer and its use as a receiving buffer is given in Figure 4.11.

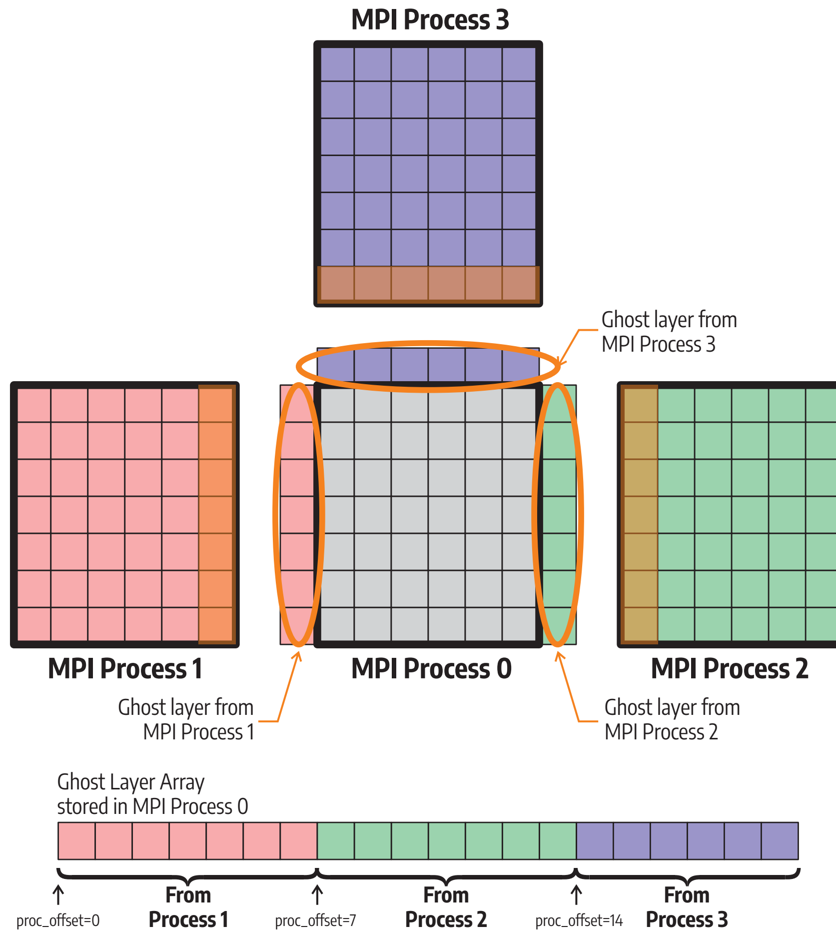


Figure 4.11: Simplified illustration of ghost layer, showing four MPI processes. On MPI process 0, the ghost layer is constructed adjacent to the outer-most local elements (i.e., left, right, and top of the sub-mesh). The bottom is the boundary, and thus, there is no need for a ghost layer. Process 0 receives data from process 1 and stores them in the ghost layer, starting from index 0 to 6. Data from processes 2 and 3 are also stored, starting from index 7 to 13 and 14 to 19, respectively.

As an example given in Figure 4.11, a ghost layer is constructed adjacent to the outermost local elements of the sub-mesh in MPI process 0. The number of ghost elements for MPI process 0 is determined by the `ghosts.elem_count` from `p4est_ghost_t`, which stores the ghost mesh structure and is initialized by calling `p4est_ghost_new` during simulation engine initialization (Figure B.7). The memory allocation for the ghost layer is performed by the user, where, in this case, the ghost layer is an array of `ElementDataBase` since the ghost elements are also represented using the same data structure as local elements.

During the ghost exchange, the MPI process 0 receives the data of the ghost elements from MPI process 1 and stores them on the ghost layer, starting from index 0 to 6. Likewise, the MPI process 0 receives the ghost data from MPI processes 2 and 3 and stores them on the ghost layer, starting from index 7 to 13 and 14 to 19. The offset in which the data is stored is given by `proc_offsets` inside the `p4est_ghost_t`. The data-receiving process is implemented as a loop over all MPI processes spawned during the simulation run. In each loop, `MPI_Irecv` is called, passing the ghost layer, the offset, the size of all ghost elements from the neighboring process, and the MPI communicator.

When running the `p4est_iterate` with `p4est_iter_face_t` callback, such as during the flux computation, the ghost layer is given as `user_data` along with the ghost mesh structure as `p4est_ghost_t`. The former holds the actual ghost element data, while the latter stores the structure of the ghost layer. The callback gives the flux kernel `p4est_iter_face_info_t`, as shown in Figure 4.8. Accessing `is.full.is_ghost` or `is.hanging.is_ghost` quickly determines whether the neighbor is a local or ghost element.

The beauty of the `p4est_iterate` is that the indexing to either an array of local elements (i.e., an array of `ElementDataBase`) or an array of ghost elements is determined automatically. This means that accessing the quadrant or octant ID through `is.full.quadid` or `is.hanging.quadid` will get the correct index to the

array of local elements if the neighbor is a local element or array of ghost elements if the neighbor is a ghost element. The flux kernel's task is to use the correct array when getting the neighboring element's data.

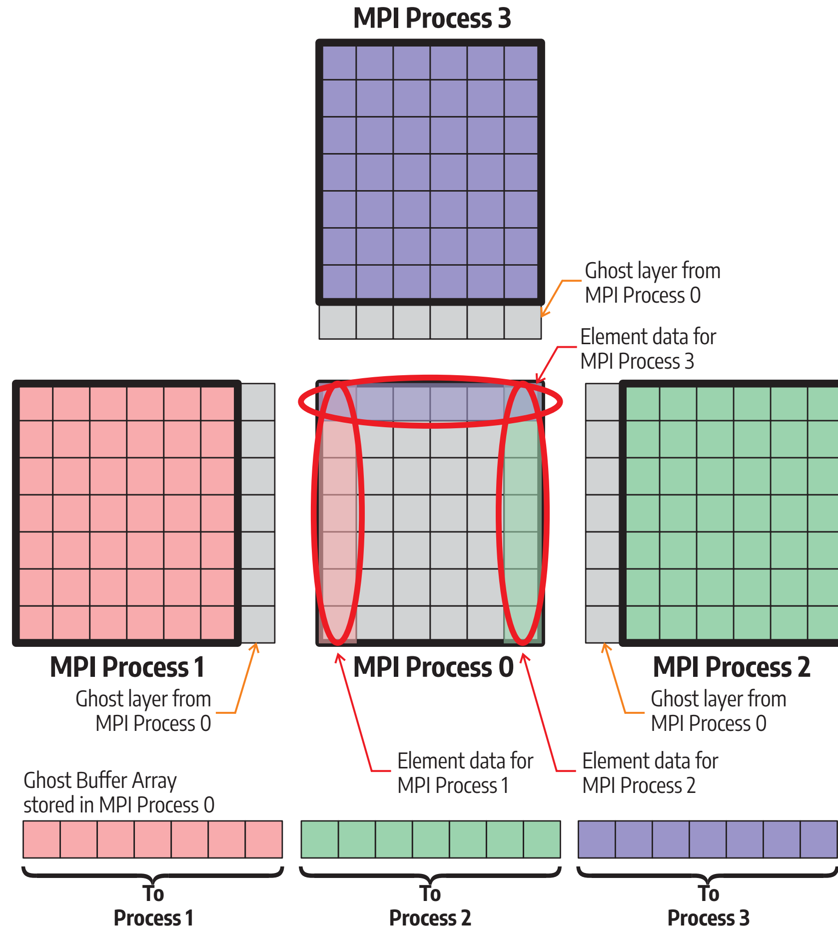


Figure 4.12: Simplified illustration of ghost buffer, showing four MPI processes. On MPI process 0, the outer-most elements are the ghost elements for other MPI processes. These elements' data are copied to the ghost buffer array before being sent into the appropriate MPI process. The two elements located on the corner of the sub-mesh are duplicated and sent twice since they are ghost elements for two processes.

4.5.1.2 Ghost Buffer

While the ghost layer is used to receive the data from other (neighboring) MPI processes, the ghost buffer is used to stage the data before sending them to other

(neighboring) MPI processes. During the ghost exchange, the MPI sends functions (i.e., `MPI_Send` for blocking or `MPI_Isend` for non-blocking) send the data inside the ghost buffer to appropriate MPI processes. A simplified illustration of the ghost buffer and its use as a sending buffer is given in Figure 4.12.

As an example given in Figure 4.12, the outermost elements on the sub-mesh held by the MPI process 0 become the ghost elements for MPI processes 1, 2, and 3. There are 18 outermost elements on MPI process 0, but there are 21 elements that need to be sent during ghost exchange. This is because some outermost elements are ghost elements for multiple MPI processes; hence, they are sent multiple times. The outermost elements are also called mirror elements by `p4est` since they are "mirrors" of the ghost layer (i.e., local elements that are ghosts in the perspective of at least one other process). The number of mirror elements for MPI process 0 is determined by the `mirrors.elem_count` from `p4est_ghost_t`. The memory allocation for the ghost buffer is performed automatically by the `p4est`.

During the ghost exchange, the `p4est` allocates the ghost buffer and copies the data of the mirror elements to the buffer. The `p4est` determines the mirror elements by accessing information provided by `mirror_proc_mirrors` inside the `p4est_ghost_t`. After copying the data of mirror elements from the local buffer (i.e., array of `p4est_quadrant_t`, see Figure 4.6) to the ghost buffer, the `p4est` sends the data inside the ghost buffer to the appropriate MPI processes. The data-sending process is implemented as a loop over all MPI processes spawned during the simulation run. In each loop, `MPI_Isend` is called, passing the ghost buffer, the total data size needs to be sent to the neighboring process, and the MPI communicator.

4.5.2 Ghost Exchange

After familiarizing with two important data structures used for ghost exchange, I discuss how the ghost exchange itself is performed using functions provided by the `p4est`. As discussed earlier, the ghost mesh structure needs to be initialized at the

beginning of the simulation. This is done by calling `p4est_ghost_new` and passing the `p4est_t` and `P4EST_CONNECT_FACE`. The `p4est_t` must have been initialized earlier by calling `p4est_new_ext`. Note that the size of the element (i.e., the size of `ElementDataBase`) is given to `p4est_new_ext` during initialization of the `p4est_t`. Figure B.7 shows the detailed high-level flow of the simulation engine initialization.

The `p4est` have various functions to perform the ghost exchange. I am mostly interested in asynchronous ghost exchange in this dissertation, as shown in simulation flow on Figure 4.9. By performing the ghost exchange asynchronously, there is a good chance to overlap parts of communication with the computation, hiding the data movement overhead. The asynchronous ghost exchange is begun by calling `p4est_ghost_exchange_data_begin` before calling the volume kernel. The `p4est_t`, `p4est_ghost_t`, and the ghost layer are given as arguments to this function. This function produces a flag, `p4est_ghost_exchange_t` that identifies the ghost exchange process. Inside this function, the non-blocking MPI functions, `MPI_Isend` and `MPI_Irecv`, are used to send the data in the ghost buffer and to receive the data to the ghost layer, respectively.

After the volume kernel finishes its execution and before calling the flux kernel, `p4est_ghost_exchange_data_end` is called, and the ghost exchange flag created earlier is given as an argument. This blocking function waits for all pending asynchronous MPI communications to finish before continuing with flux computation. This mechanism is vital to avoid computing flux before the data of the ghost elements are fully synchronized. While on paper, the asynchronous ghost exchange is great for overlapping computation with communication to hide the communication overhead, the asynchronous ghost exchange's effectiveness depends on the used MPI library. This will be discussed in more detail in Section 5.6.3.

4.6 Result Verification and Performance Measurement

Finally, the diagnostic and report are given at the end of the simulation, as shown in Figure 4.9. This report helps analyze the simulation's behavior and whether there is any issue with its kernel implementations and configurations. The output of the diagnostic and report at the end of the simulation is given in Figures B.3 and B.4. Additional information on diagnostic and statistic output is given in Appendix B.2.2.

4.6.1 Numerical Accuracy

The numerical accuracy is determined by computing the L_2 error, which is the sum of the squared error between the final (computed) solutions and the analytical solutions. The derivation of the analytical solutions has been discussed thoroughly by Kaufman and Levshin (2005), and has been used in many studies, such as by Wilcox et al. (2010); Appelö and Hagstrom (2018); Chan (2018); Feng et al. (2007),

The kernel `compute_L2_error` is used to calculate the analytical solution and integrate (accumulate) the squared error. This kernel is given to `p4est_iterate` as `p4est_iter_volume_t` callback, and thus the L_2 error is computed for each element. The resulting L_2 error for each element is then accumulated to get the local L_2 error. For multi-CPU runs, the local L_2 error for each sub-mesh (i.e., MPI process) needs to be gathered. This is accomplished using MPI function `MPI_Reduce` with operation `MPI_SUM`.

4.6.2 Solution Ranges

In addition to the L_2 error, the diagnostic and report produce the range of solutions for each variable. This helps determine whether the simulations have numerical instability, causing the simulations to blow up. The kernel `update_solution_min_max` is used to find the minimum and maximum values for each variable. This kernel is given to `p4est_iterate` as `p4est_iter_volume_t` callback, and thus the minimum and maximum values for each variable are computed for each element. The result-

ing minimum and maximum values are then compared against all elements within the local sub-mesh to get the local minimum and maximum values. The local minimum and maximum values for each sub-mesh (i.e., MPI process) must be compared for multi-CPU runs. This is accomplished using the MPI function `MPI_Reduce` with operation `MPI_MAX` and `MPI_MIN` to find the global maximum and minimum values, respectively.

4.6.3 Runtime Performance

The runtime performance of the simulation represents the time needed to run the simulation kernels. It gives the breakdown of the time spent in each kernel (i.e., volume, flux, integration) and the time spent in ghost exchanges. The time is measured using the wall clock through `MPI_Wtime`. This runtime performance is used to compare the performance of the CPU implementation with the GPU implementation (Chapters 5 and 6) and the PIM implementation (Chapter 7).

Chapter 5: Accelerating dG-based Wave Simulations using GPUs

In this chapter[1], I describe the first contribution of this dissertation: accelerating the dG-based wave simulations using Graphics Processing Units (GPUs). The work done in this chapter has been published as a paper with the title *GAPS: GPU-Acceleration of PDE solvers for Wave Simulation* by Hanindhito et al. (2022).

The baseline codes (i.e., the codes that will be accelerated using GPU) are the CPU codes I have explained in Chapter 4. Aside from the implementation described in this dissertation, many frameworks use multi-core CPUs to solve wave equations, such as the approaches by Poursartip et al. (2020); Wilcox et al. (2010). However, CPUs may not be the most efficient hardware for wave simulations. The sophisticated features[2] occupying most of the silicon area of the CPU may be beneficial for general applications with complex execution flow and irregular data access. However, these features may not benefit wave simulations, which have considerable predictability, regularity, and parallelism. Instead, having many simpler cores with more silicon area dedicated to computation (i.e., ALUs) is more beneficial (Section 2.6.2). With its vast number of processing cores, the graphics processing unit (GPU) is a promising candidate to accelerate wave simulations.

Porting the CPU codes to GPU codes and getting satisfactory performance is not straightforward; it is a lengthy process to get the most performance out of the

[1]Contents of this chapter have been published and appear in:
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 Proceedings of the 36th ACM International Conference on Supercomputing (Virtual Event) (ICS '22). Association for Computing Machinery, New York, NY, USA, Article 30, 13 pages. https://doi.org/10.1145/3524059.3532373.
I proposed, designed, implemented, and evaluated the main ideas while collaborating with the co-authors on writing the manuscript.

[2]e.g., out-of-order execution engines, branch predictors

GPU while maintaining numerical accuracy and stability. First, I have to modify the data structures used in the CPU codes to be more friendly for the GPU to efficiently perform massively parallel operations (Section 5.1). Then, I modify the execution flow to perform computation at the interior nodes of the elements and at the face nodes of the elements (Section 5.2), eliminating the need for using serial methods of `p4est` to iterate through elements and nodes.

After taking care of data structure and execution flow, I began writing the GPU simulation kernels, which mostly follow the CPU codes. The basic GPU implementation is done at this point, and its performance and numerical accuracy are evaluated using methods described in Section 3.1.3. Using `nvprof` profiling tool, I found that data movement is the key bottleneck of the basic GPU implementation. Thus, I devise several optimization strategies to reduce the data movement overhead (Section 5.4). In addition, I also perform a study on the effect of element order on the simulation performance (Section 5.5.4). Before wrapping up the single-GPU implementation, I perform the performance analysis of the code, including the simulation time, speed-up over CPU, and roofline (Section 5.5).

Finally, the last part of the puzzle is to make the GPU codes scalable to handle larger problem sizes by utilizing multi-GPUs on multiple compute nodes. I describe how to modify the ghost exchange of `p4est` to run on GPU, utilize NVIDIA GPUDirect technology, perform ghost exchange optimization to reduce the volume of data being exchanged, and compare various MPI libraries with and without asynchronous progression support (Section 5.6). All these efforts allow the GPU code to achieve near-perfect weak scaling over 128 GPUs.

5.1 Data Structure Modification for GPU

The first step in developing GPU codes is transforming the data structure to be more GPU-friendly, extracting high-level parallelism efficiently. While `p4est` works beautifully for handling the mesh on CPU code, it does not have built-in support

for running on GPU. Instead of writing the AMR library from scratch, my approach is to extend the functionality of `p4est` by developing functions and data structures optimized for GPU execution. Previous work that uses this strategy includes the work by Fernando et al. (2022), which extends the open-source Dendro-gr library to work on GPUs. The Dendro-gr library is open-source, state-of-the-art AMR Numerical Relativity codes by Fernando et al. (2019); Fernando and Sundar (2020); paralab (2021).

Although `p4est_iterate` is highly versatile to iterate through the elements' interior, face, and corner while taking care of ghost elements for face and corner, its serial nature makes it difficult to extract parallelism efficiently on GPU. Figure 4.7 shows how `p4est_iterate` is used for volume computation (Section 4.4.2). Each thread (i.e., MPI process) running on one CPU core iterates through the subset of the mesh by calling the `p4est_iterate`, which calls `compute_volume` (i.e., volume kernel) as callback function passed as `p4est_iter_volume_t` argument for each element in the subset.

This means each thread processes each element on its subset mesh *serially*, each with calls to the volume kernel. The fact that each element is independent of the other and can be processed in parallel during volume computation (i.e., element-level parallelism) is not properly utilized by `p4est_iterate` to get better performance. Moreover, nodes within an element can also be processed in parallel (i.e., node-level parallelism), which cannot be done by `p4est_iterate` since one thread handles the mesh subset. Therefore, the two sources of parallelism are not fully leveraged (Section 4.1.4).

Finally, multiple pointer operations are performed on `p4est_mesh_t` to get access to the element's user data on the array inside `p4est_tree`, on which the `compute_volume` is performed. These pointer operations are inefficient for GPU. A simple data structure, such as a contiguous array, is preferable for GPU to achieve higher parallel execution efficiency.

5.1.1 Mesh Data

The first modification to the data structure is to simplify how the elements are stored in the memory. The original implementation from `p4est` uses an array of `p4est_quadrant_t` implemented as `sc_array_t` inside `p4est_tree`. The `sc_array_t` is a struct containing an array of characters, providing versatility to store any data and metadata, which includes the size and number of elements. The `sc_array_t` is defined by `libsc`.

For the GPU version, I use a basic contiguous array of elements while maintaining the indexing to each element inside the array according to the numbering system used by `p4est`. In other words, if the element's data is represented by a struct called `ElementDataBase` (Section 4.2.3), then the array to store it will be an array of `ElementDataBase`. This array is copied from CPU memory to GPU memory at the beginning of the simulation. Then, the GPU kernel is launched on the GPU to run operations on this array. This array is copied back to the CPU memory at the end of the simulation or during the checkpoint to dump the mesh data for visualization purposes (Appendix B.2.3).

This data structure modification greatly simplifies extracting parallelism at both the element- and node-level (Sections 5.3.1, 5.3.2, and 5.3.4). Multiple threads can be summoned to process each element simultaneously, and each element can have many dedicated threads that process each node concurrently. In other words, instead of only using one thread to handle all elements in the mesh subset, each element can have a group of threads. Simple direct access to each element and each node within the element improves the efficiency of parallel execution on the GPU.

5.1.2 Mesh Structure

While volume computation enjoys the simple data structure for storing the data of the elements, mesh structure information is still required for flux computation. In `p4est`, the built-in function `p4est_iterate` automatically traverses the

mesh structure when computing flux, as shown in Figure 4.8. The call back function given as `p4est_iter_face_t` argument calls for every group of elements facing each other. Then, `p4est_iter_face_info_t` is passed to the callback function, where the information of each element can be found, including the quadrant ID, whether the neighboring element is full or hanging, and whether the neighboring element is local (i.e., stored in the same memory as the primary element) or ghost element (i.e., stored in other memory). Note that, for multi-CPU and multi-GPU implementation, the flux computation is performed after the ghost exchange has been done (Sections 4.5 and 5.6).

Facing the same problem with the volume computation, `p4est_iterate` for flux computation is serial in nature, processing one group of elements facing each other at a time. To efficiently extract as much parallelism as possible in GPU, efficient methods must be implemented to allow each thread to find neighboring elements quickly instead of traversing through the mesh structure. Two approaches were developed: the iterative method and the look-up table (LUT) method. Using either method, each thread can quickly determine neighboring elements simultaneously. The basic GPU implementation (Section 5.3) uses the iterative method for finding the neighboring elements, while the optimized GPU implementation (Section 5.4) will use the LUT-based method.

5.1.2.1 Iterative Method

For conforming mesh with no hanging elements (i.e., all elements are the same size), an iterative method can be used to determine the neighboring elements, as shown in Algorithm 3. The algorithm finds the neighboring element on a particular face of the primary element[3]. It expects that dimension ($\mathcal{D}$)[4], refinement level ($\mathcal{R}_L$, see Section 4.1.1), and the number of refinement children ($\mathcal{R}_C$)[5] are properly defined.

[3]The algorithm must be run for each face (i.e., four times for 2D space, six times for 3D space).
[4]i.e., $\mathcal{D} = 2$ for 2D space, $\mathcal{D} = 3$ for 3D space
[5]i.e., $\mathcal{R}_C = 4$ for 2D space, $\mathcal{R}_C = 8$ for 3D space

The inputs to the algorithm are the primary quadrant ID ($\mathcal{Q}_1$)[6], the axis ID ($\mathcal{I}$)[7], and the normal vector of the face of interest ($\mathcal{N}$)[8] of the primary element. It outputs the neighboring quadrant ID at the end of the iteration ($\mathcal{Q}_2$). Note that if $\mathcal{Q}_2 = -1$, the primary element does not have a neighboring element on that particular face, which happens when the primary element is located at the boundary.

The algorithm starts with initialization to the internal variables: $\mathcal{Q}_2$, $\overline{\mathcal{Q}_2}$, $\mathcal{I}_f$[9], $\mathcal{S}$[10], and $\mathcal{A}$[11]. The rest of the algorithm is inside a loop whose iteration depends on the refinement level $\mathcal{R}_L$. First, $\mathcal{C}$ and $\overline{\mathcal{Q}_2}$ are calculated by using mod and div to $\mathcal{R}_C$, respectively. Then, the coordinate of the primary element ($\mathcal{P}_P$) along the axis $\mathcal{I}$ is computed. Next, the neighboring element coordinate ($\mathcal{P}_N$) is calculated using the information of the primary element's coordinate and the normal vector ($\mathcal{N}$) of the face of the primary element. If $\mathcal{P}_N$ falls within $\{0, 1\}$, then the neighboring element quadrant ID $\mathcal{Q}_2$ can be computed since it is located at the same (tree) level as the primary element (See Figure 4.6). Otherwise, the algorithm must go to the next iteration to find the correct (tree) level by memorizing $\mathcal{A}$ and updating $\mathcal{S}$.

5.1.2.2 Look-up Table (LUT) Method

In addition to supporting only conforming mesh, the iterative method is sub-optimal since the time complexity of determining the neighboring element is $\mathcal{O}(\mathcal{R}_L)$, meaning it depends on the refinement level being used. Therefore, look-up tables

[6] i.e., the index to the array of quadrants, following the `p4est` numbering scheme

[7] This is the axis where the two neighboring faces are located at, which parallels to the normal vector of the faces. For x, y, and z axes, the value are $\mathcal{I} = 0$, $\mathcal{I} = 1$, and $\mathcal{I} = 2$, respectively.

[8] This is the normal vector from the perspective of the face of the primary elements. The value is $\mathcal{N} = -1$ if it points to the negative direction of the axis and $\mathcal{N} == +1$ if it points to the positive direction of the axis).

[9] $\mathcal{I}_f$ is used for multiplier or divisor of step $\mathcal{S}$ with value of $\mathcal{I}_f = 1$, $\mathcal{I}_f = 2$, and $\mathcal{I}_f = 4$ for x, y, and y axis, respectively.

[10] This represents the step of increment of quadrant ID for each refinement level. For example, in 3D space, the value of $\mathcal{S}$ will equal to 1, 8, 64, and 512 for refinement level 0, 1, 2, and 3, respectively. Likewise, in 2D space, the value of $\mathcal{S}$ will equal to 1, 4, 16, and 64 for refinement level 0, 1, 2, and 3, respectively.

[11] This is used for accumulating the result of $\mathcal{S} \times \mathcal{I}_f$ for previous refinement level.

Algorithm 3: Find Neighboring Elements using Iterative Method

Defined:

Dimension $\mathcal{D}$, Refinement level $\mathcal{R}_L$, Refinement children $\mathcal{R}_C$

Inputs:

Element Quadrant ID $\mathcal{Q}_1$, Axis ID $\mathcal{I}$, and Normal Vector $\mathcal{N}$

▷ $\mathcal{I} = \{0, 1, 2\}$, $\mathcal{N} = \{-1, +1\}$

Outputs:

Neighbor Element Quadrant ID $\mathcal{Q}_2$

Initialization:

1 $\mathcal{Q}_2 \leftarrow -1$ ▷ Initialized to -1 (no neighbor)

2 $\overline{\mathcal{Q}_2} \leftarrow \mathcal{Q}_1$ ▷ Temporary variable $\overline{\mathcal{Q}_2}$

3 $\mathcal{I}_f \leftarrow 1 \ll \mathcal{I}$ ▷ Axis factor, $\mathcal{I}_f$

4 $\mathcal{S} \leftarrow 1$ ▷ Quadrant ID step

5 $\mathcal{A} \leftarrow 0$ ▷ Quadrant ID accumulator

Main:

6 **for** *all* **j** *in* $\{0, ..., \mathcal{R}_L - 1\}$ **do**

7 $\quad \mathcal{C} \leftarrow \overline{\mathcal{Q}_2} \bmod \mathcal{R}_C$

8 $\quad \overline{\mathcal{Q}_2} \leftarrow \overline{\mathcal{Q}_2} \operatorname{div} \mathcal{R}_C$

9 $\quad \mathcal{P}_P \leftarrow (\mathcal{C} \operatorname{div} \mathcal{I}_f) \bmod 2$

10 $\quad \mathcal{P}_N \leftarrow \mathcal{P}_P + \mathcal{N}$

11 $\quad$ **if** *(*$\mathcal{P}_N = 0$ or $\mathcal{P}_N = 1$*)* and $\mathcal{Q}_2 = -1$ **then**

12 $\quad\quad \mathcal{Q}_2 \leftarrow \mathcal{Q}_1 + \mathcal{N} \times (\mathcal{S} \times \mathcal{I}_f - \mathcal{A})$

13 $\quad$ **end**

14 $\quad \mathcal{A} \leftarrow \mathcal{A} + \mathcal{S} \times \mathcal{I}_f$

15 $\quad \mathcal{S} \leftarrow \mathcal{S} \ll \mathcal{D}$

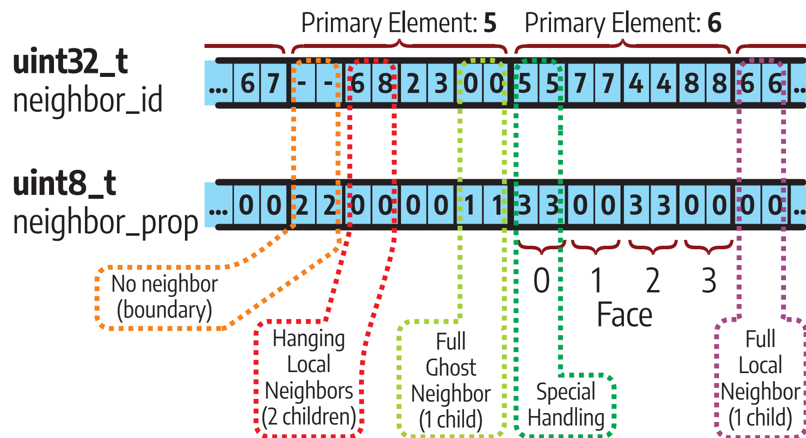
16 **end**

(LUTs) are implemented using two contiguous arrays consisting of one array of 32-bit integers (`neighbor_id`) and one array of 8-bit integers (`neighbor_prop`), as shown in Figure 5.1. With these LUTs, the time complexity of finding neighboring elements is $\mathcal{O}(1)$ at the expense of small memory for storing the LUTs[12]. The size of the LUT depends on the problem's dimension and whether there are any hanging elements. For example, in 3D space problems with hanging elements, each element has 6 faces, 4 children (neighboring) elements per face, for a total of 24 children per element. Since the mesh structure is not changed throughout the simulation run, the LUTs are constructed once at the beginning on CPU, using `p4est_iterate` with special function passing through `p4est_iter_face_t` callback (Figure 4.8). Then, the LUTs are copied to GPU memory just before the simulation loop begins.

The first array, `neighbor_id`, stores the quadrant ID of the neighboring elements in unsigned 32-bit integers, allowing to keep track of 4 billion neighboring elements. This number may not be reachable in a realistic setting since each GPU has limited off-chip memory capacity. If the neighbor element is stored on the same memory as the primary element (i.e., local element), the quadrant ID is the index to the array of quadrants stored in GPU memory. On the other hand, if the neighbor element is stored on another GPU's memory (i.e., ghost element), the quadrant ID is the index to the array of ghosts.

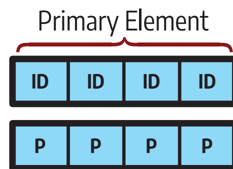
The second array, `neighbor_prop`, stores the properties of neighboring elements in 8-bit integers. If the value is 0, the neighboring element is local; if the value is 1, the neighboring element is ghost. If the element is located at the boundary and does not have a neighboring element at that face, the value will be 2. There is a special case for the hanging element where the smaller element faces the larger element,

[12]It needs 114 MB to store the LUT for 1 million elements in 3D space with support for hanging elements. In this case, the size of each single-precision element in acoustic and elastic problems is 32 KB and 62 KB, respectively. This means the LUT memory space is only 0.36% and 0.19% of the total problem sizes. The proportion can be lower for problems without hanging elements or problems with double-precision elements

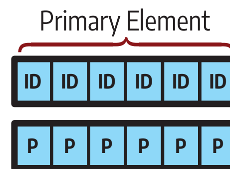


LUT Configurations

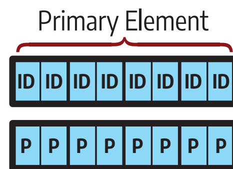
2D Space without hanging
(4 faces, 1 child per face)



3D Space without hanging
(6 faces, 1 child per face)



2D Space with hanging
(4 faces, 2 children per face)



3D Space with hanging
(6 faces, 4 children per face)

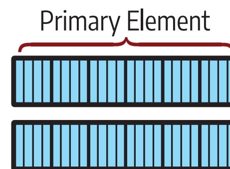


Figure 5.1: Look-up Table (LUT) for finding neighboring elements, consisting of one 32-bit integer array for storing neighbor elements' quadrant ID and one 8-bit integer array for storing neighbor elements' properties.

which requires special handling. Instead of starting with the smaller element, the larger element must be found first, and the computation can be performed alongside its children. A value of 3 indicates this special case.

5.2 Execution Flow Modification for GPU

After modifying both mesh structure and mesh data, the execution flow of the kernels must be modified to run on GPU to allow for as much parallelism extraction as possible. With the modification to these data structures, the execution flow will no longer depend on the serialized `p4est_iterate` method to iterate through elements and pairs of faces of the elements neighboring each other. As with the CPU code, there are two scenarios: iterating through the interior (`iter_volume`) and the face of elements (`iter_face`).

5.2.1 Execution Flow for Interior of Elements

Modifying the execution flow for iterating through the interior of elements is done by eliminating the call to `p4est_iterate`. Instead of giving the kernel as `p4est_iter_volume_t` callback, the kernel is launched on GPU directly. Note that the mesh structure and data must have been copied to the GPU memory before launching any GPU kernels. Figure 5.2 shows the modification for execution flow in GPU at a high level. Compared to the original execution flow in CPU shown in Figure 4.7, the flow is simplified, without any pointers operation to access each element's data. This execution flow is used for simulation kernels operating on the elements' interior: volume kernel, mass-inverse kernel, and integration kernel.

As with the CPU flow, the CPU thread, denoted by blue, operates on a mesh subset. If only one processor is used, it operates on a whole mesh. In GPU execution flow, the CPU thread copies the mesh data and mesh structure from CPU to GPU memory. It is also responsible for copying back the result from GPU memory to CPU memory. Note that the data transfer is only done once at the beginning and once

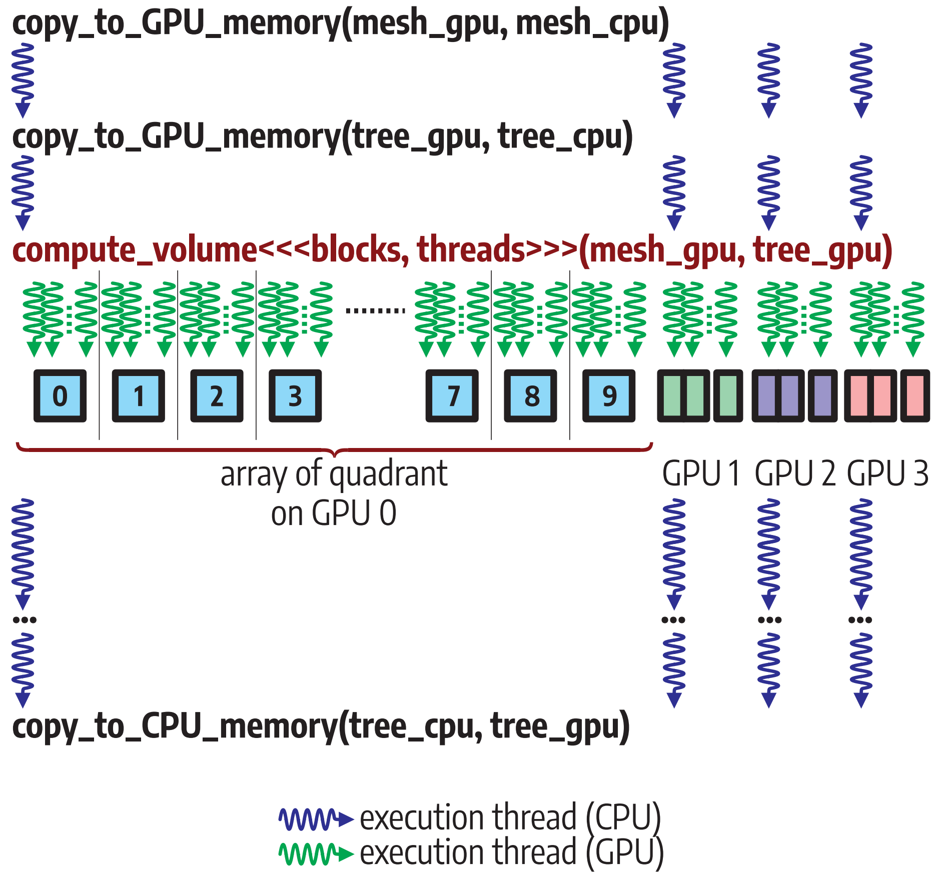


Figure 5.2: The serial nature of `p4est_iterate` is not suitable for GPU's massive parallel execution. Instead of processing (local) elements one at a time, each group of threads handles one element and performs volume computation simultaneously. Storing the element's data in a contiguous array ensures each thread can efficiently access the data of each element given to them.

at the end of the simulation. Then, the CPU thread is responsible for launching the simulation kernel on the GPU. In this way, one CPU thread handles one GPU, which is carried over to the multi-GPU implementation, discussed later in Section 5.6.

The decorator <<<blocks,threads>>> is used as a GPU kernel launch parameter. It defines the number of thread blocks and the number of threads per thread block (Section 2.6.2.3). Usually, the total number of threads across all thread blocks equals the number of parallelisms that can be extracted. Based on this launch parameter, GPU threads, denoted by green, are spawned to run the codes defined in the kernel. Here, one element (i.e., quadrant) has several threads dedicated to processing each node simultaneously. For example, if the number of nodes inside an element is 512, then the ideal number of threads per element is 512, dedicating one thread to process one node. In this way, every node on every element can be processed in parallel, extracting both element-level and node-level parallelism.

5.2.2 Execution Flow for Face of Elements

The iteration over pairs of faces of the neighboring elements has a more complex execution flow than the iteration over the interior of elements since it needs to determine the neighboring elements. This execution flow is used for the flux kernel, with the modified flow for GPU illustrated in Figure 5.3. Compared to the CPU execution flow shown in Figure 4.8, the GPU flow is more straightforward with eliminated pointer operations to determine the neighboring elements. It does not rely on `p4est_iterate` with `p4est_iter_face_t` callback to launch the kernel; instead, the kernel is launched on GPU directly, assuming the mesh data and mesh structure are already in GPU memory. The GPU kernel responsible for ghost exchange must be called before executing the flux kernel in multi-GPU implementation, as discussed later in Section 5.6. Otherwise, the kernel can be launched directly following the volume kernel in single-GPU implementation, as shown in Figure 4.9.

The first step of the execution for each thread is to get the neighboring ele-

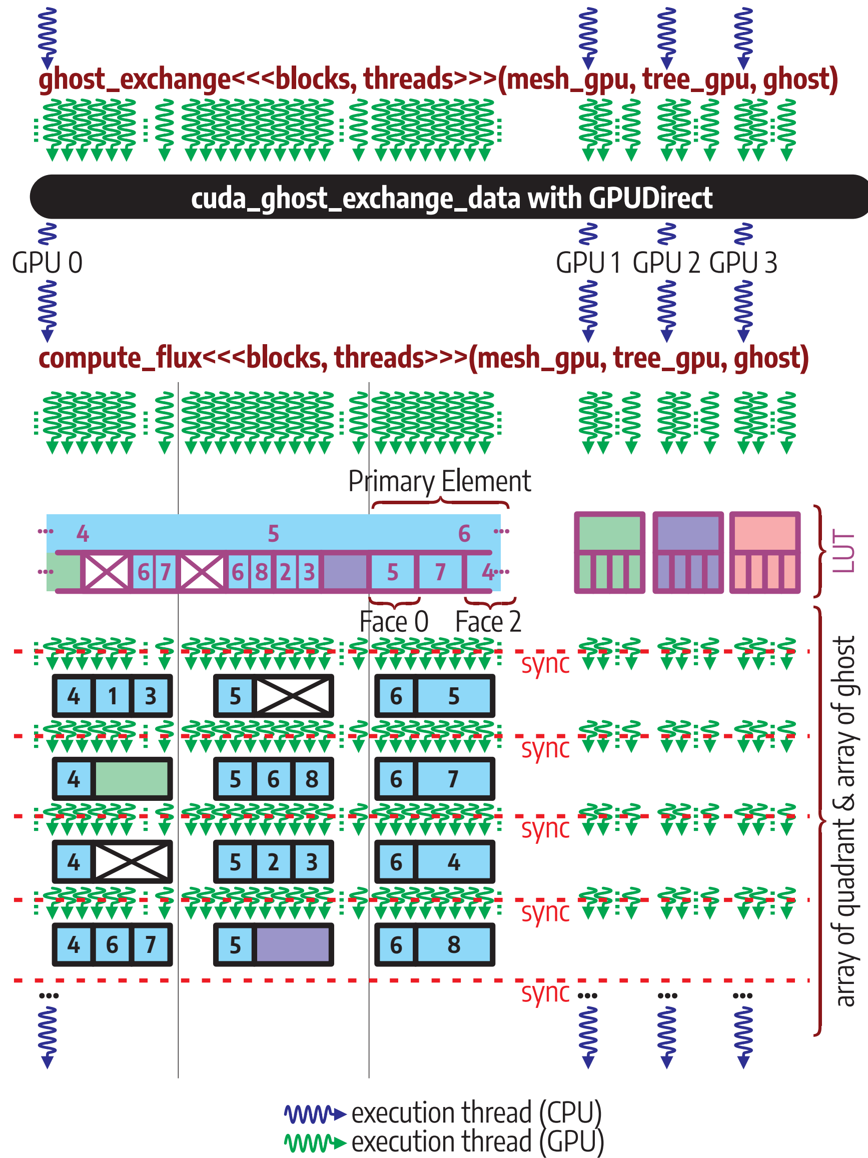


Figure 5.3: The serial nature of `p4est_iterate` is not suitable for GPU's massive parallel execution. Instead of processing a group of elements one at a time, each group of threads handles a group of elements and performs flux computation simultaneously, with some handling to race conditions. However, finding the neighboring elements through pointer operations is inefficient; instead, each group of threads accesses the look-up table (LUT) to determine the neighboring element quickly. This LUT contains the index to an array of quadrants or an array of ghost elements, allowing each thread to efficiently access the data of each group of elements given to them.

ments. Instead of relying on `p4est_iter_face_info_t` and subsequent pointer operations, each thread can directly obtain the neighbor information, either by using the iterative method (Algorithm 3) or by look-up table method (Figure 5.1). Then, the flux computation is performed.

Due to the data hazard discussed later in Section 5.3.3.1, each face's flux computation is done serially. There should be a synchronization barrier between flux computation for each face of the element, either by launching a separate kernel for each face (i.e., implicit device-wide synchronization barrier) or by using `__syncthreads()` (i.e., explicit thread-block-wide synchronization barrier). The former is used for basic flux kernel implementation, discussed in Section 5.3.3, while the latter is used for optimized flux kernel implementation, discussed in Section 5.4.1. For example, in a 3D problem, since each element has six faces, there should be six serialized flux computations, one for each face.

Although the GPU execution flow for the flux kernel is more simplified in terms of the number of branches and pointer operations, some branches cannot be avoided. For example, not all nodes within an element need flux computation, and not all faces have neighboring faces. The latter necessitates two branches for computing flux vector: the flux vector when the neighbor's face is present and the boundary flux vector when there is no neighbor's face, as seen in the detailed high-level execution flow of flux kernel in Figure B.10.

5.3 Basic GPU Simulation Kernels Implementation

The development of basic GPU kernels generally follows the CPU kernels, described in Section 4.4. However, some basic optimizations are performed to allow extracting as much parallelism as possible, keeping the cores inside the GPU busy. During the development, the simulation result from the GPU code is compared against the analytical solution and the result from the CPU code to ensure its numerical accuracy, as discussed in Section 3.1.3.

The GPU simulation follows the CPU's flow, as illustrated in Figure 4.9, albeit with few modifications. At the beginning of the simulation, `p4est` is still used for generating, partitioning, and distributing mesh. Then, the data for each element (i.e., `p4est_tree` is copied and stored as a contiguous array inside the GPU memory (i.e., array of quadrants). If LUT-based neighbor search is used (Figure 5.1), the LUTs are generated using the provided call back function to `p4est_iterate`. The LUTs are then copied to the GPU memory. Then, the inner and outer loops of the simulations are executed as usual, except instead of calling CPU functions, they call GPU kernels. At the checkpoint (e.g., to dump data for visualization; see Appendix B.2.3) and at the end of the simulation, the updated data for each element is copied back to the CPU memory. The basic implementation of each GPU kernel and different opportunities to parallelize the operation are described below.

5.3.1 Mass-Inverse Kernel

As already described in Section 4.4.1, the mass inverse kernel is only called once at the beginning before entering the simulation loop. This kernel is straightforward and follows the execution flow for the element's interior, as illustrated in Figure 5.2. The total number of threads performing mass-inverse computation is $N_{element} \times N_{nodes_per_element}$. Since the kernel is very short and only launched once, it is not considered for optimizations due to its negligible impact on overall simulation performance, especially when it runs for thousands of time steps.

5.3.2 Volume Kernel

The volume computation is an entirely local operation and the most compute-intensive simulation kernel. However, it is highly parallelizable: each node within an element and each element within the mesh can be computed independently. The parallelism is extracted at both the node and element levels by assigning one thread to handle the volume computation of one node. It follows the element's interior

execution flow as illustrated in Figure 5.2. The total number of threads performing volume computation is $N_{element} \times N_{nodes_per_element}$. Several basic optimizations are applied to the volume kernel on GPU, as explained below.

5.3.2.1 Precomputing Derivative of Shape Functions

In the CPU code, the computation of velocity divergence, pressure gradient (for acoustic problem), and stress divergence (for elastic problem) involves the computation of the derivative of shape functions, done repeatedly for every node and every simulation time step. Since the values are constant, instead of recomputing the same values repeatedly, they can be precomputed and stored inside special memory inside the GPU called constant memory (Figure 2.8).

For 3D space with 512-node elements, only 64 single-precision values or double-precision values must be stored inside the constant memory for a total of 256 bytes or 512 bytes, respectively. This significantly reduces the amount of computation needed and, at the same time, utilizes GPU's fast, constant memory to improve performance. Note that the Gauss-Lobatto-Legendre (GLL) integration points are also stored inside the GPU's constant memory, which consists of eight nodal values and eight weight values for a total of 64 bytes (single-precision) or 128 bytes (double-precision).

5.3.2.2 Reducing Branches

Branch operations incur additional overhead for execution in GPU, especially if the outcome of the branches is not the same for threads within a warp. Threads inside a warp execute in lock-step fashion. Any differences in the execution path reduce execution efficiency since the threads will be serialized (i.e., executed one-by-one for each execution path), as explained in Section 2.6.2. Multiple loops are merged during the computation of velocity divergence, pressure gradient (for acoustic problem), and stress divergence (for elastic problem) into one giant loop to reduce branch and compare operations. Note that the loop over each node within an element

shown in Figure B.9 does not exist in GPU code since the parallelism is extracted at the node level; instead of looping over each element using only single thread, each node is processed by a dedicated thread simultaneously.

5.3.3 Flux Kernel

Unlike volume computation, flux computation is a non-local operation since it requires data from neighboring elements. Although the computation itself is not as intense as the volume kernel, many branch operations potentially degrade the overall execution performance on GPUs, as described by Wu et al. (2013). The branch operations are needed to determine the neighboring elements and to check whether an element is a boundary element. If an element is a boundary element, it will have a special treatment to compute the boundary conditions, as shown in Figure B.10. In addition, the iterative method (Algorithm 3) is used to determine neighboring elements. Three challenges for flux computation on GPUs are as follows.

5.3.3.1 Data Hazard

Race conditions can potentially happen for nodes located at the edge or the corner of an element since these nodes are computed for two (edge) and three (corner) faces. With the effort of extracting as much parallelism as possible, multiple threads may access and update the data of these nodes simultaneously, causing incorrect results. Therefore, flux computation at each face must be serialized to avoid these data hazards by computing the flux in one direction at a time. For 2D and 3D space, there are four and six separate flux computation kernel launches corresponding to the number of faces of each element. This mechanism is implemented on the CPU thread as `for` loop across faces, as shown in Algorithm 4. Therefore, considering branch and divergence overhead, the potential parallelism will be less than $N_{element} \times N_{nodes_per_face}$.

5.3.3.2 Double Computation

Due to extracting parallelism at the element level, there is a potential for double flux computation. For example, two threads may handle the flux computation of the same pair of faces with different self-elements (i.e., the self-element of one of the threads is the neighboring element of the other thread). Therefore, a flag mechanism is needed to indicate whether the flux computation has been node. To simplify the implementation and avoid storing more information in memory, quadrant ID can be used as a flag; if an element has a higher ID value than the neighboring element, then the thread handling this element is responsible for computing the flux on the neighboring element. Thread handling the neighboring element will not perform any flux computation. This flag mechanism is implemented as `if` conditional on GPU thread, comparing $\mathcal{Q}_s$ and $\mathcal{Q}_n$ as shown in Algorithm 4.

Algorithm 4: The algorithm to run the basic flux kernel on GPU

Defined:
Dimension $\mathcal{D}$, face ID $\mathcal{F}$
Self quadrant ID $\mathcal{Q}_s$, Neighbor quadrant ID $\mathcal{Q}_n$

Main:
CPU Thread:

```
1  for F in {0, ..., 2 × D} do
2      Launch Flux Kernel for face F

       GPU Thread (inside the flux kernel):
3      Get Neighbor's information
4      if Neighbor is present then
5          if Q_s ≥ Q_n then
6              Compute flux vector at face F.
7          end
8      else
9          Compute boundary flux vector at face F.
10     end
11 end
```

5.3.3.3 Node Look-up

Only nodes on the elements' faces, corners, and edges are considered during flux computation. Although flux computation is serialized across faces, node-level parallelism for all nodes within the same face must be done to improve performance. However, the index information for nodes located on the face must correctly access its data inside the `variables` array of `ElementDataBase`. The CPU code uses branches to determine the face nodes' numbering scheme and map it to the correct node index, which is inefficient for GPU. An iterative method is developed to determine the node index more efficiently, as shown in Algorithm 5.

The algorithm expects that the problem dimension $\mathcal{D}$ and the number of nodes in each direction $\mathcal{M}$[13] are properly defined. The inputs to the algorithm are the face ID $\mathcal{F}$[14] and the node ordinal $\mathcal{O}$[15]. It outputs the correct node index according to the node numbering scheme of `p4est`. The algorithm starts with the initialization of the internal variables: $\mathcal{A}$[16], $\mathcal{P}$[17], $\mathcal{S}$[18], $\mathcal{N}$, and $\overline{\mathcal{O}}$. The rest of the algorithm is inside a loop whose iteration depends on the problem dimension $\mathcal{D}$. In each iteration, one conditional branch is used inside the loop to properly increment the node index $\mathcal{N}$. The optimized version of fused volume and flux kernel (Section 5.4.1) uses a lookup table instead of an iterative method to determine the node index, as shown in Figure 5.5.

---

[13] In 3D space with 512-node elements, $\mathcal{M} = 8$.

[14] The face ID corresponds to the face numbering scheme of `p4est`.

[15] This is the sequence of nodes on the particular face. For example, in 3D space with 512-node elements, each face has node sequence $O = \{0..64\}$

[16] The axis where the face is located; $\mathcal{A} = 0$, $\mathcal{A} = 1$, and $\mathcal{A} = 2$ for x, y, and z axis, respectively.

[17] The normal vector of the face; $\mathcal{P} = 0$ if normal vector points toward negative value and $\mathcal{P} = 1$ if normal vector points toward positive value

[18] Multiplier for indexing to the note; updated every iteration.

Algorithm 5: Find Face Node Index using Iterative Method

Defined:
Dimension $\mathcal{D}$ and number of nodes in each direction $\mathcal{M}$

Inputs:
Face ID $\mathcal{F}$ and Node ordinal $\mathcal{O}$

Outputs:
Node index according to element numbering scheme $\mathcal{N}$

Initialization:

1 $\mathcal{A} \leftarrow \mathcal{F}$ div 2 ▷ Axis where the face located.
2 $\mathcal{P} \leftarrow \mathcal{F}$ mod 2 ▷ The pointing direction of the face.
3 $\mathcal{S} \leftarrow 1$ ▷ Multiplier based on axis
4 $\mathcal{N} \leftarrow 0$ ▷ Node index to find
5 $\overline{\mathcal{O}} \leftarrow \mathcal{O}$ ▷ Temporary node ordinal

Main:

6 **for** *all* **j** *in* $\{0, ..., \mathcal{D}\}$ **do**
7   **if** $\mathbf{j} \neq \mathcal{A}$ **then**
8         $\mathcal{C} \leftarrow \overline{\mathcal{O}}$ mod $\mathcal{M}$
9         $\overline{\mathcal{O}} \leftarrow \overline{\mathcal{O}}$ div $\mathcal{M}$
10         $\mathcal{N} \leftarrow \mathcal{N} + \mathcal{C} \times \mathcal{S}$
11   **else**
12         $\mathcal{C} \leftarrow \mathcal{C}$
13         $\overline{\mathcal{O}} \leftarrow \overline{\mathcal{O}}$
14         $\mathcal{N} \leftarrow \mathcal{N} + \mathcal{P} \times (\mathcal{M} - 1) \times \mathcal{S}$
15   **end**
16     $\mathcal{S} \leftarrow \mathcal{S} \times \mathcal{M}$
17 **end**

5.3.4 Integration Kernel

The integration kernel is the shortest kernel that computes the LSRK4 time integration (Section 2.3.3). Memory accesses dominate this kernel, and since the kernel is very short, there is little room for improvement. It follows the execution flow for the element's interior. Since the variables on each node can be processed independently, the total number of threads performing integration is $N_{element} \times N_{nodes_per_element} \times N_{variables}$. The Runge Kutta Coefficients are stored inside the GPU's constant memory for fast access.

5.4 GPU Simulation Kernels Optimizations

The simulation kernels described in Section 5.3 are the basic implementation of GPU kernels. Although they offer significant performance improvements over the CPU version (Section 5.5.2), thanks to a higher degree of parallelism being extracted, they still have room for further improvements. As described by Wulf and McKee (1995), the memory wall and the limited capacity of on-chip memory on GPU makes data movement between off-chip (i.e., DRAM) and on-chip memory (i.e., SRAM) quickly become the performance bottleneck, even for GPUs equipped with high-bandwidth memory (HBM) capable of providing Terabytes per second bandwidth. It is also shown by Imani et al. (2019a, 2020). Therefore, in this section, three strategies were developed for optimizing the performance of GPU simulation kernels, particularly in reducing data movement overhead.

5.4.1 Kernel Fusion

Launching kernels on GPUs incurs overhead, which may not be negligible in time-marching scheme simulation, where the kernels are launched multiple times for every time step. In addition, the state of the GPU is reset every time the kernel finishes its execution, including the contents of registers and caches. This means that the subsequent kernel needs to bring back the data from off-chip to on-chip memory

if it wants to perform operations on the data processed by the previous kernel.

As described in earlier work by Wang et al. (2016a); Wahib and Maruyama (2014), kernel fusion tries to merge two or more GPU kernels into one. By fusing several consecutive kernels, the data is most likely preserved inside the registers and caches throughout the kernel execution, helping to minimize the data movement overhead by eliminating the need to fetch the data from off-chip memory. Based on the data flow of the simulation, the possible candidates are volume and flux kernels. Due to the data hazard explained in Section 6.2.2, the integration kernel is best kept as a separate kernel; the integration must only be run once all volume and flux contributions for every element have been calculated. One can use a synchronization barrier to ensure the integration is run after all threads finish computing volume and flux. However, hardware support for explicit GPU-wide synchronization is expensive. Instead, implicit GPU-wide synchronization through separate kernel launches can be used since the GPU will wait for the previous kernel to conclude its execution before launching a new kernel using the same execution stream.

Before fusing volume and flux kernel, multi-GPU implementation (Section 5.6) that relies on ghost exchange for flux computation must be considered. In other words, the fused kernel of volume and flux cannot be launched until the ghost exchange is completed since, during the flux computation, an element may have ghost elements as its neighbor. However, as illustrated in Figure 4.9, overlapping the ghost exchange with computation as much as possible is desired to hide the communication overhead.

The flux kernel is separated into internal and external flux kernels to better overlap ghost exchange with computation, as shown in Figure 5.4. The former does not need a ghost exchange to finish and can be fused with volume computation, while the latter is delayed and launched as a separate kernel once the ghost exchange is completed. This separation also gives more time to hide the communication overhead of ghost exchange with volume and internal flux computation. Leveraging the LUT-based neighbor search (Figure 5.1), the flux computation that requires ghost elements

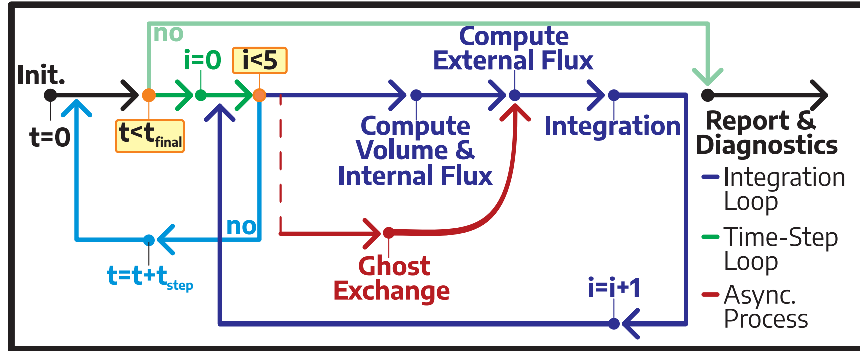


Figure 5.4: The modification to the simulation flow by fusing the volume kernel and flux kernel as an effort of optimization to reduce data movement and kernel launch overhead. To facilitate multi-GPU implementation (Section 5.6), the flux kernel is separated into two parts: internal flux, where all of the elements are located at the same GPU, and external flux, where neighboring elements are located at different GPUs. The volume and internal flux kernel can be fused and executed without waiting for ghost exchange to finish, while the external flux does.

can be easily skipped during internal flux computation. On the other hand, during external flux computation, the kernel only takes care of all nodes with ghost elements as their neighbors.

Using the fused volume and internal flux kernel, the total number of threads performing volume and flux computation is $N_{element} \times N_{nodes_per_element}$. This means one thread handles one node throughout the kernel execution. Instead of using Algorithm 5, LUT-based node index (Figure 5.5) is used by each thread to determine whether it handles the node located on the face or not. Each thread can easily check whether a node is located on a particular face by binary right-shifting the lookup value with the face ID, which is a very fast operation in hardware. For example, if a node has a lookup value of 00010101, then $(00010101 \gg 4) \wedge 1 = 1$ is true since this node is located at face 4. On the other hand, $(00010101 \gg 5) \wedge 1 = 1$ is false since this node is not located at face 5.

However, during the computation of internal flux, thread divergence is inevitable. Each thread that handles node with node-lookup equal to 0 (i.e., 00000000)

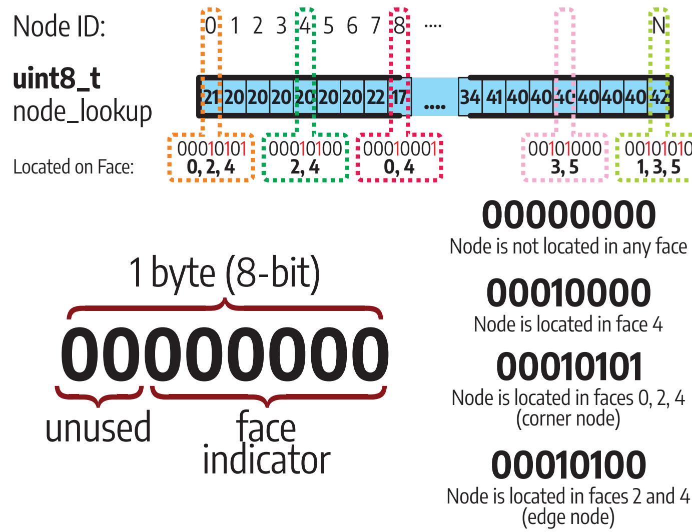


Figure 5.5: Look-up Table (LUT) for finding node index according to the numbering scheme of `p4est`, consisting of one 8-bit integer array. The first 4 bits (2D space) or 6 bits (3D space) indicate the node location on the face while the other bits are unused.

does not need to perform flux computation since the node is not located on any faces. On the other hand, each thread that handles a node with node-lookup not equal to 0 needs to perform flux computation one, two (i.e., edge node), or three times (i.e., corner node) depending on the node's location. In a 512-node element, 216 nodes do not need flux computation as they are not located on the face, while the 296 nodes require flux computation, including 8 corner nodes (three flux computation) and 72 edge nodes (two flux computation). The external flux kernel also faces the same divergence problems with an addition of skipping all face nodes that do not need a ghost element (i.e., flux already computed during internal flux computation).

5.4.2 Shared Memory Utilization

In addition to registers and caches, which are hardware-managed, GPU features another type of on-chip memory called shared memory that lets the user self-manage its usage, as discussed in Figure 2.9. The shared memory is available at limited capacity to each Streaming Multiprocessor (SM) inside the GPU, allowing

threads within the same thread block to share data, as described by Wu et al. (2015). It is allocated when the kernel starts and reset when execution finishes.

From Volta architecture, NVIDIA GPU features unified memory consisting of L1 cache and shared memory sharing the same on-chip memory structure, as described by NVIDIA Corporation (2017); NVIDIA Corporation (2020a). If the kernel is not using shared memory, the whole unified memory can be used as a hardware-managed L1 cache. This means using shared memory will reduce the on-chip memory capacity available to the L1 cache, potentially hurting performance if the advantages of using shared memory do not outweigh the performance loss from having a smaller L1 cache. Therefore, shared memory must be carefully managed; choosing the correct data to store is crucial for performance.

Based on how often data is accessed, either read or write, two potential candidates can be stored in shared memory: `contributions` and `variables`. The `contributions` are written repeatedly during volume and flux computation, and since volume and internal flux are fused into one kernel (Section 5.4.1), storing `contributions` inside shared memory guarantees them to stay inside the SM throughout the kernel execution. On the other hand, the `variables` are read repeatedly during volume and flux computation. For the same reason as `contributions`, keeping them close to the SM is better, reducing the need to fetch them from off-chip memory. However, storing both `contributions` and `variables` in shared memory is not the best idea since the capacity of shared memory is limited and is shared with the L1 cache, potentially degrading performance when the unified memory available for the L1 cache is too small.

After investigating the memory access pattern for `variables`, it is found that storing `variables` inside the shared memory may not yield improved performance, primarily when the threads handling one element spread across multiple thread blocks with their own shared memory space. Therefore, only `contributions` are stored inside the shared memory. Modifying existing code to use the shared memory for

`contributions` is almost straightforward; instead of writing to the `contributions` array inside each element, the `contributions` are written to the array stored inside the shared memory. At the end of the fused volume and (internal) flux kernel execution, the `contributions` are copied to the array inside each element. This way, instead of writing multiple times[19] to off-chip memory during kernel execution, only one memory write is done at the end.

5.4.3 Register Allocation Improvement

Another room for improvement is to control the register allocation to improve SM occupancy. SM occupancy measures how many warps can be scheduled inside an SM at a given time. Four factors determine SM occupancy: warps per SM, thread block per SM, register per SM, and shared memory per SM. Having multiple warps occupy one SM allows the scheduler to aggressively switch to another while waiting for the previous warp to finish its memory access, hiding memory access latency and improving overall performance, as discussed in Section 2.6.2. With only one warp, the SM is stalled since it waits for the thread block to finish the memory access.

To better support more warps per SM, each thread's register allocation is controlled by instructing the compiler, `nvcc`, to allocate registers accordingly using decorator `__launch_bounds__`. This enables the SM to schedule more than one warp since there are sufficient registers to hold more threads. However, limiting per-thread register allocation may cause some register spills; some data spills over the local memory due to insufficient registers, as shown by Rawat et al. (2018). Fortunately, a few register spills would not hurt performance, but the code is refactored to reduce the number of spills.

[19] i.e., one write for volume contribution and up to three reads and three writes for flux contributions since the node can be located at the edge or the corner of an element, necessitating multiple flux computations.

5.5 Performance Analysis

In this section, the performance achieved by the GPU is compared against the CPU. Based on the implementation discussed in Sections 5.3 and 5.4, there are three versions of GPU codes: `GPU_base`, `GPU_fl`, and `GPU_fls`, as shown in Table 5.1. The `GPU_base` implements volume kernel described in Section 5.3.2 and flux kernel described in Section 5.3.3 with iterative method for both neighbor look-up (Algorithm 3) and node index look-up (Algorithm 5). On the other hand, `GPU_fl` implements fused volume and flux kernel as described in Section 5.4.1 along with LUT-based neighbor search (Figure 5.1) and LUT-based node index look-up (Figure 5.5). Finally, the `GPU_fls` modifies `GPU_fl` to utilize shared memory (Section 5.4.2) and improve register allocation (Section 5.4.3). At this point, all of the GPU code versions can only utilize single GPU, while the CPU code (Chapter 4) already has support for utilizing multiple CPUs.

Kernel Set Name	Optimization Strategy	Described in Section
`GPU_base`	Basic (`base`)	5.3
`GPU_fl`	Fused Volume and Flux with LUT-based Node Look-up (`fl`)	5.4.1
`GPU_fls`	Fused Volume and Flux with LUT-based Node Look-up (`fl`); Shared memory utilization for `contributions` and improved register allocation (`s`)	5.4.1, 5.4.2, and 5.4.3

Table 5.1: The GPU kernel configuration flavors for performance evaluation with different optimization levels.

5.5.1 Simulation Runtime

Figure 5.6 shows the per-kernel breakdown of simulation time in acoustic, elastic with Riemann flux solver, and elastic with Central flux solver problems for three GPU code flavors: `GPU_base`, `GPU_fl`, and `GPU_fls`. Each problem runs on 3D space with refinement-level 5 consisting of 32,768 elements in either single-precision (FP32) or double-precision (FP64). The overhead time, shown in orange, is the time spent initializing and finishing the simulation, which includes generating the look-up tables, copying the mesh structure and data from CPU to GPU memory at the

beginning of the simulation, and copying back the mesh data from GPU to CPU memory at the end of the simulation.

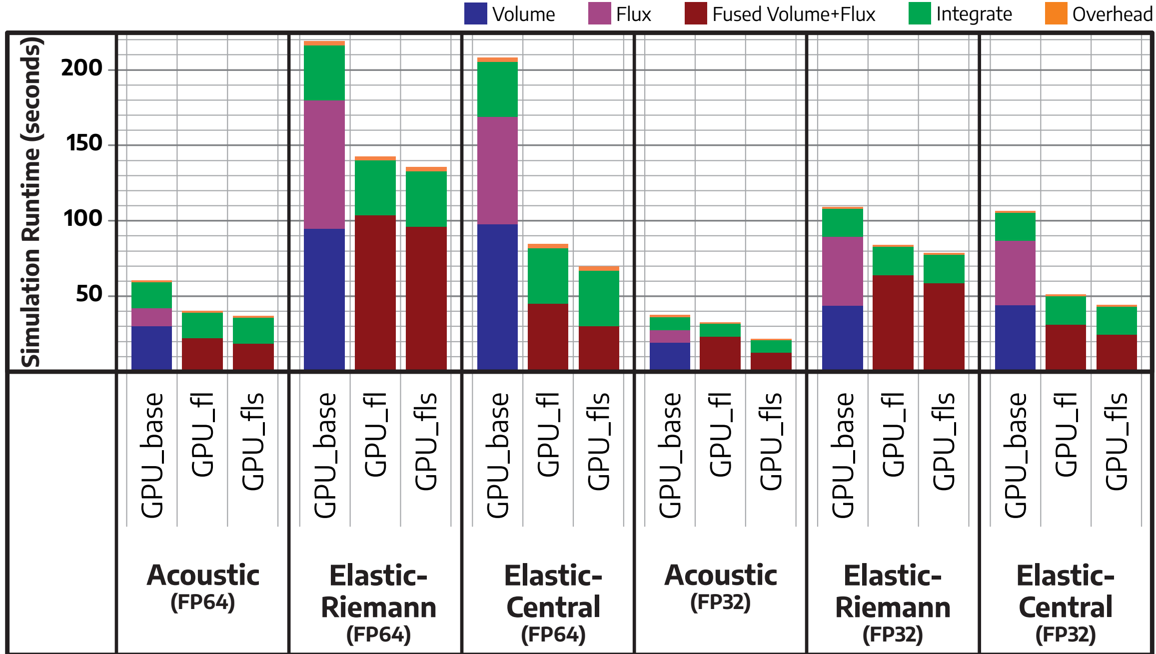


Figure 5.6: GPU simulation time and the total execution time for each kernel for 3D space at refinement level 5 (32,768 elements) running on single NVIDIA Tesla V100 GPU for 1000 time steps. There are three flavors of GPU code corresponding to the optimization efforts: `GPU_base`, `GPU_fl`, and `GPU_fls`. The former uses separate volume (dark blue) and flux (purple) kernels, while the other uses fused volume and flux kernels (dark-red). The integration (green) kernel is the same across different flavors. The overhead is the additional time for preparing the simulation (e.g., constructing LUT, copying mesh data from CPU to GPU memory) and finishing the simulation (e.g., copying back the mesh data from GPU to CPU memory).

For `GPU_base` with the acoustic problem, most simulation time is spent on computing volume (dark-blue): 49.8% and 51.5% of total simulation time for FP64 and FP32, respectively. This is expected as volume is the most compute-intensive part of the simulation, even though parallelism is already extracted as much as possible. The integration is the second-longest runtime kernel, consuming 28.2% and 23.3% of total simulation time for FP64 and FP32, respectively. Since the integration kernel is very short, there is no room for optimization (Section 5.3.4), and thus the three flavors

of GPU code share the same integration kernel. Finally, the flux kernel consumes 20% and 21.4% of total simulation time for FP64 and FP32, respectively. The FP32 simulation is 1.6×faster than the FP64 thanks to reduced data size that needs to be fetched from off-chip memory, which becomes apparent when discussing the roofline analysis on Section 5.5.3.

Moving to the elastic problem with `GPU_base`, it can be seen the proportion of time spent on flux computation increases significantly. Unlike acoustic problems, in elastic problems, flux computation is the second-longest runtime kernel, consuming 40.7% and 34% of total simulation time for Riemann and Central flux solvers, respectively, in FP64. In FP32 arithmetic, it consumes 41.6% and 40.0% of total simulation time for Riemann and Central flux solvers, respectively. Finally, it is worth noting that the Riemann flux solver consumes more time than the Central flux solver: 1.13× and 1.04× longer in FP64 and FP32, respectively.

In both `GPU_fl` and `GPU_fls` flavors, both volume and flux kernels are fused together (dark-red). Fusing these two kernels reduces the kernel launch overhead and data movement overhead associated with bringing the data from off-chip memory for subsequent kernels. For all problems in FP64 and FP32, the time spent to execute the fused volume and flux kernel is significantly less than the total time spent to execute separate volume and flux kernels, like in `GPU_base`. In the acoustic problem, the fused volume and flux kernel in `GPU_fl` is 1.89× and 1.18× faster than separate volume and flux kernels in FP64 and FP32, respectively. In the elastic problem with the Riemann flux solver, the fused volume and flux kernel are 1.73× and 1.39× faster. The most significant improvement in runtime is enjoyed by the elastic problem with the Central flux solver: 3.73× and 2.77× faster in FP64 and FP32, respectively.

Finally, `GPU_fls` further shorten the runtime of fused volume and flux kernel of `GPU_fl`. With shared memory and improved register allocation, it is 1.19× and 1.85× faster for acoustic problems in FP64 and FP32, respectively. For elastic with Central flux solver, it is 1.49× and 1.28× faster in FP64 and FP32, respectively. However,

for elastic with Riemann flux solver, the improvement is insignificant for the reason that will be discussed in Section 5.5.3.

5.5.2 Speed-up Over CPUs

After discussing the simulation runtime and how optimization plays an important role in improving overall simulation performance, the speed-up of GPU against the CPU is analyzed. First, the baseline for comparison must be established. In the experiments, the CPU codes were run on two types of CPU: IBM POWER9 (Section 3.2.2) and Intel Xeon Platinum 8160 (Section 3.2.1). The IBM platform features two sockets CPU, each with 20 cores for a total of 40 cores, while the Intel platform features two sockets CPU, each with 24 cores for a total of 48 cores. The CPU codes were run using Spectrum MPI (IBM) and Intel MPI (Intel) to utilize all available CPU cores. The communication overhead accounts for an average of 9.5% and 17% of total simulation time for IBM and Intel platforms, respectively.

Although both MPI implementation supports asynchronous progression (Section 5.6.3), it was not enabled since it leads to overall performance degradation due to oversubscription[20]: 9.3% and 73.5% longer simulation times for the IBM and Intel platforms, respectively. Since the Intel platform has an average speed-up of 1.36× compared to the IBM platform for handling 32,768 elements, as shown in Figure 5.7, the IBM platform is used as the baseline.

Figure 5.8 shows the speed-up of GPU codes against the CPU baseline. On average, the speed-up achieved by the GPU over baseline CPU is 45.67×, 69.30×, and 84.15× for `GPU_base`, `GPU_fl`, and `GPU_fls`, respectively. The acoustic problem enjoys

[20]Oversubscription occurs when the number of threads being executed exceeds the available CPU cores. When asynchronous progress is enabled, Spectrum MPI and Intel MPI use an additional helper thread for each MPI process to handle communication. This incurs additional OS overhead of thread context switching and core contentions that negate the benefit of asynchronous progression and lead to performance degradation. GPU codes do not suffer the same problem since the primary computation is offloaded to the GPU, leaving most CPU cores available to handle asynchronous progress threads.

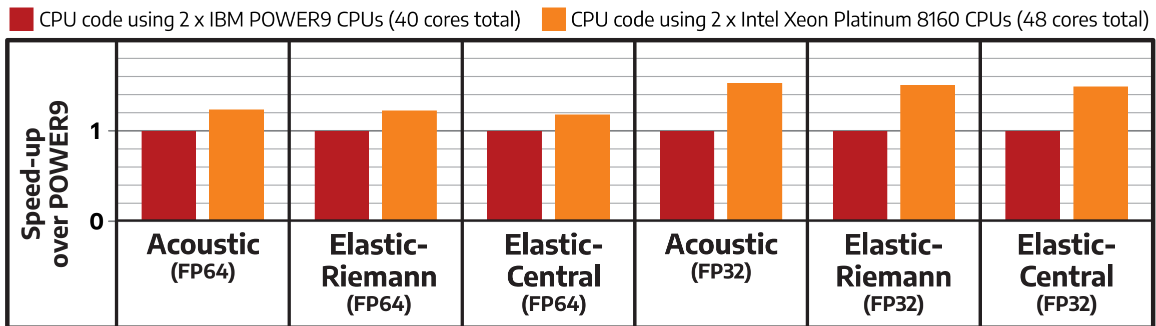


Figure 5.7: Speed-up of the CPU code running on two sockets of Intel Xeon Platinum 8160 CPUs for a total of 48 cores (orange) compared to the baseline CPU (red), consisting of two sockets of IBM POWER9 CPUs for a total of 40 cores.

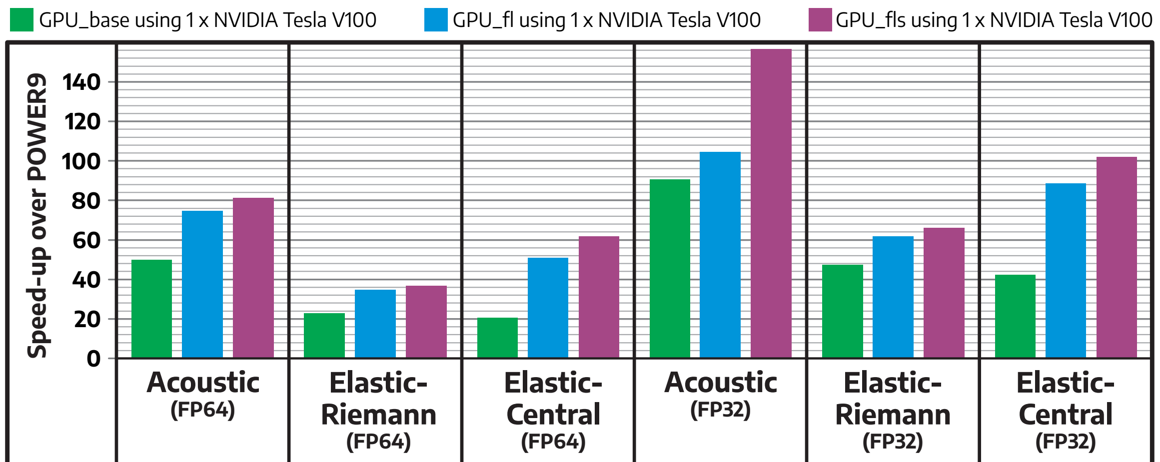


Figure 5.8: Speed-up of three different GPU code flavors on single NVIDIA Tesla V100 GPU compared to the baseline CPU, consisting of two sockets of IBM POWER9 CPUs for a total of 40 cores.

the most significant speed-up: 49.83×, 74.67×, and 81.32× in FP64 for `GPU_base`, `GPU_fl`, and `GPU_fls`, respectively. The speed-up is even higher for FP32: 90.68×, 104.54×, and 156.55×, respectively. The acoustic problem has relatively short kernels and fewer intermediate results, allowing more warps to be scheduled in each SM to hide memory access latency with minimal register spills. On the other hand, the elastic problem with the Riemann flux solver enjoys the least speed-up: 22.83×, 34.89×, and 36.87× in FP64 or 47.49×, 61.89×, and 66.03× in FP32 for `GPU_base`, `GPU_fl`, and `GPU_fls`, respectively. The Riemann flux solver is a relatively long kernel with many intermediate results, making scheduling multiple warps in each SM challenging. In addition, limiting the register usage per thread causes more register spills that degrade performance.

In summary, the kernel fusion (Section 5.4.1), LUT-based neighbor search (Figure 5.1), and LUT-based node index search (Figure 5.5) brings an average of 1.74× speed-up on `GPU_fl` over the `GPU_base`. Meanwhile, using shared memory and improving the register allocation on `GPU_fls` brings an additional average speed-up of 1.17×. As discussed next on Section 5.5.3, the overall simulation is memory-bounded, and thus having these optimizations (Section 5.4) helps reduce data movement and memory pressure, improving overall performance.

5.5.3 Roofline Analysis

In this section, each GPU kernel is further characterized with the help of a roofline analysis, as described in Section 2.5.2. Figure 5.9 shows the roofline chart for FP64 (top) and FP32 (bottom). From the chart, it can be observed that the integration kernel is a memory-bound kernel as it is hitting the slope at the left position of the chart (i.e., low arithmetic intensity). Since it is a very short kernel, there is no room for optimization; thus, it is left as it is. The flux kernel is located near the slope, albeit not hitting it, which indicates it is a memory-bound kernel with its execution inefficiency due to many branch operations. Finally, the volume kernel

is located more on the right side of the slope. However, it is still a memory-bound kernel, although it has the highest arithmetic intensity compared to the other kernels.

Fusing the volume and flux improves the arithmetic intensity and compute throughput: the fused kernel is located at the top and right of its original position in the roofline model. This translates to the 1.74× average speed-up discussed in Section 5.5.2. Further optimization with shared memory brings the fused kernel to the top, albeit slightly to the left. This means it has higher compute throughput but slightly lower arithmetic intensity due to increased data movement from reduced L1 cache capacity. Nevertheless, this translates to 1.17× average speed-up over fused kernel without shared memory.

5.5.4 Effects of Element Order on Performance

As discussed in Section 4.1.2 and Appendix A.3.3, high-order elements (i.e., elements with higher number of nodes) are favored in many wave simulations, due to their ability to limit the dispersion error. It is preferred to have a smaller number of higher-order elements compared to a larger number of lower-order elements from a hardware perspective since, due to dG, the former has a better locality. This section investigates this claim by having two element configurations: mesh with 64-node elements and mesh with 512-node elements, labeled as `N64` and `N512`, respectively. The total number of nodes from all elements within the mesh is equal for both cases, at approximately 63 million. This means the `N64` mesh has eight times more elements than `N512`. The wave simulation is run for 1000 time steps using `GPU_fls` kernels on NVIDIA Tesla V100 GPU.

Figure 5.10 shows `N512` mesh is, on average, 1.12x faster than `N64` to run the wave simulations. The `N512` element has 216 out of its 512 nodes located in the interior of the element, subjected only to volume computation, which has high arithmetic intensity and no inter-element communications. The rest of the nodes (57.8%) are subjected to flux computation. On the other hand, the `N512` element has

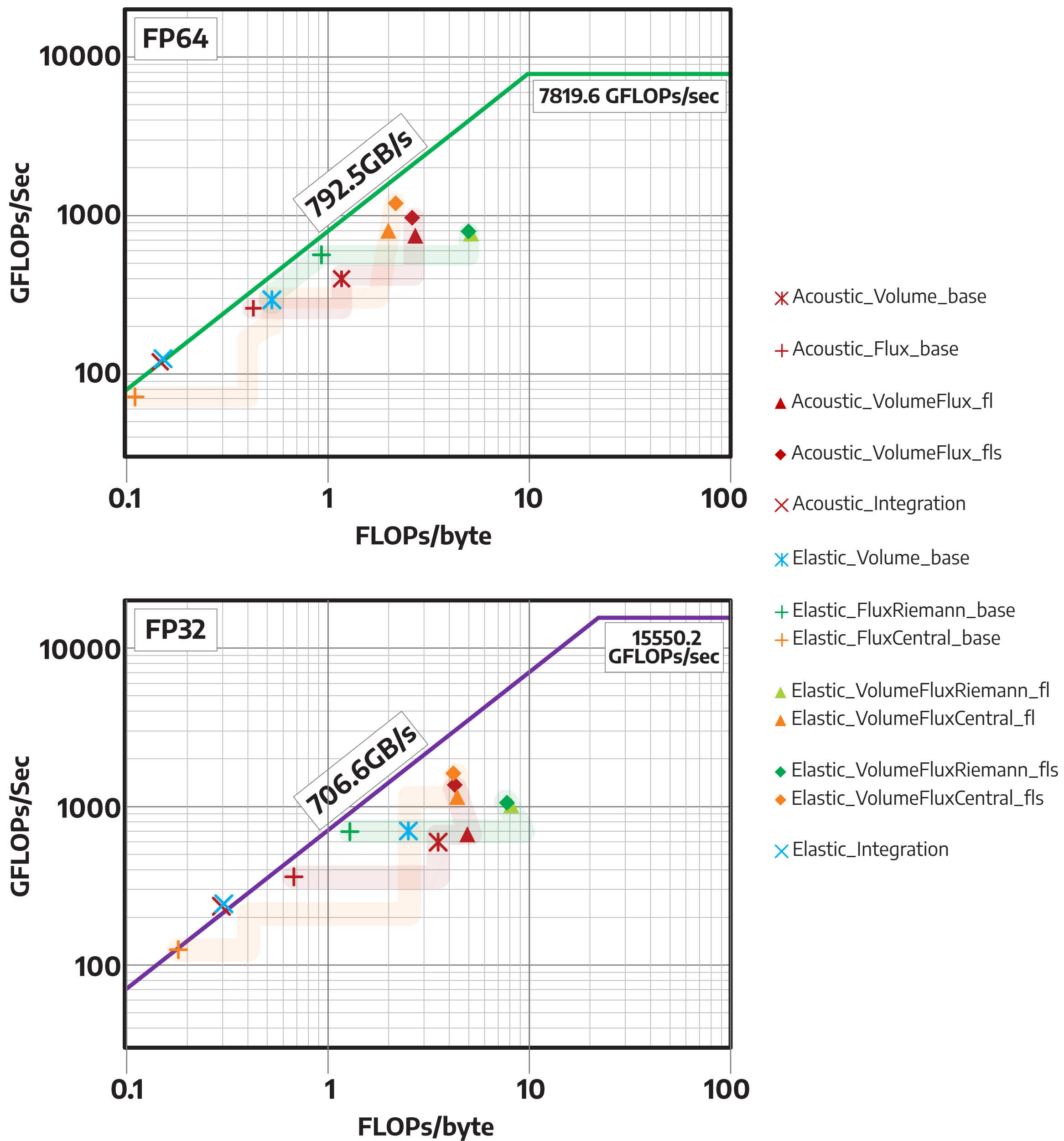


Figure 5.9: The roofline analysis for all GPU kernels shows optimizations' effect for double-precision (left) and single-precision (right) simulation runs. The roofline model of NVIDIA Tesla V100 GPU is obtained using Empirical Roofline Tool (ERT) by Yang (2015). The colored path indicates the change in the characteristics of each kernel after optimization: red for acoustic kernels, green for elastic-riemann kernels, and orange for elastic-central kernels.

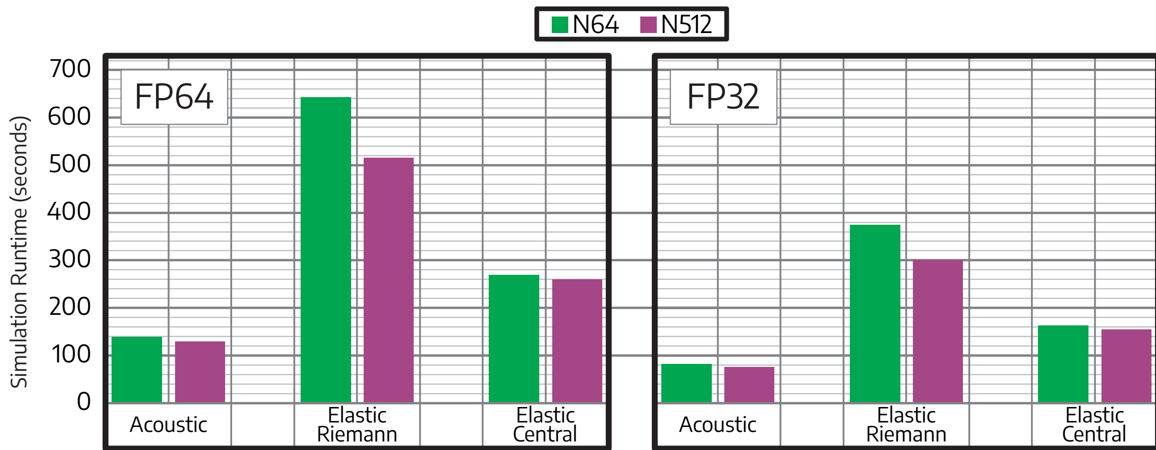


Figure 5.10: The performance of wave simulation using a mesh with 64-node elements (N64) and 512-node elements (N512) on double precision (left) and single precision (right) arithmetic. The N512 has more local operations than N64, improving overall simulation performance.

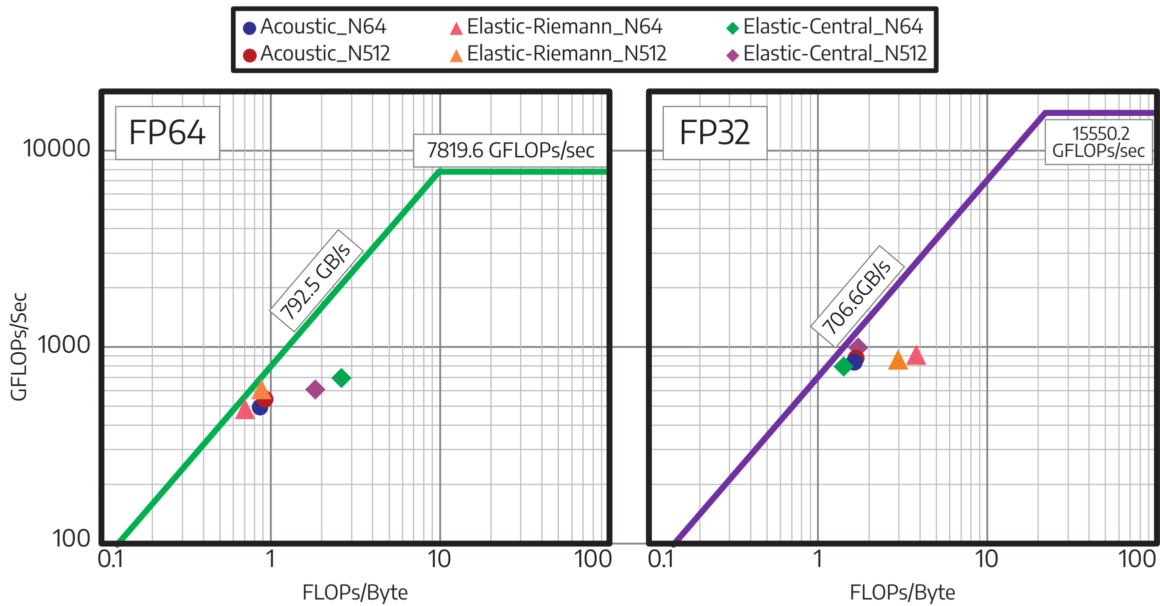


Figure 5.11: The roofline analysis for investigating the character of wave simulations when using a mesh with 64-node elements (N64) and 512-node elements (N512) on double precision (left) and single precision (right) arithmetics. Generally, the 512-node mesh results in higher arithmetic intensity and compute throughput than the 64-node mesh. However, in some cases, the 64-node element easily fits into single thread block, eliminating the need for spreading the computation into multiple thread blocks running on multiple SMs, resulting in a better locality.

only 8 out of its 64 nodes located in the element's interior; 87.5% of the nodes are subjected to flux computation. Therefore, the non-local to local operations ratio on `N64` is higher than `N512` mesh. It has been discussed in Section 5.3.3 that the flux kernel has low execution efficiency due to the branch operations. In addition, the `N64` has eight times more elements than the `N512`, significantly adding the number of flux computations that must be performed. Therefore, it is confirmed that having high-order elements is preferable from a hardware perspective.

Figure 5.11 shows the roofline analysis of the acoustic and elastic wave simulations when configured to run with `N64` and `N512` meshes. The roofline chart plots the application's behavior, not the individual kernels. By observing the application's behavior on the roofline chart, it can be implied that, generally, the `N512` mesh delivers higher arithmetic intensity and compute throughput than the `N64` mesh. In some cases, the `N64` has higher arithmetic intensity since handling one element using one thread block containing 64 threads is possible. Using 64 threads in one thread block eases the register allocations, reducing register spilling and improving the locality since the element can be stored inside one SM, instead of spreading them into multiple SM, just like what Section 5.4.3 did by limiting the number of threads per thread blocks (i.e., no longer at 512 threads per thread block) to allow multiple thread blocks scheduled into one SM. Further optimizations, discussed in Chapter 6, aim to improve the locality of processing single 512-node element using one thread block, keeping it inside single SM on GPU.

5.5.5 Early Performance Exploration on Newer GPU Architecture

This section describes the early performance exploration of running the GPU implementation of the wave simulation on newer GPU architecture, the NVIDIA A100 GPU. The run was performed in one compute node on Lonestar6 (Section 3.2.3). No code modification was performed, and the codes were compiled with `nvcc` targeting the Ampere (A100) architecture instead of Volta (V100). Figure 5.12 shows the performance achieved for acoustic and elastic wave simulations running `GPU_fls` kernels

on refinement level 5 problem with 32,768 elements.

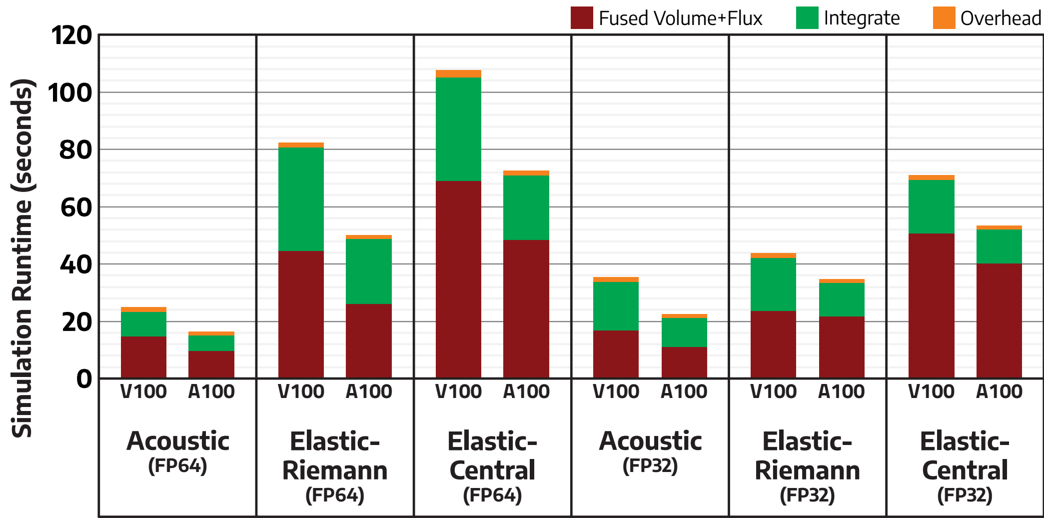


Figure 5.12: The simulation performance on NVIDIA Tesla V100 and the newer NVIDIA A100 GPUs, running for refinement level 5 with 32,768 elements and `GPU_fls` kernels. On average, A100 achieved 1.47×speed-up over V100.

The A100 achieved an average speed-up of 1.47× over V100. The A100 has 1.39× peak throughput on double and single precision arithmetic over V100. In addition to the higher peak throughput, the performance improvement is primarily due to the increased L1 cache (1.5× larger) and the increased L2 cache (6.67× larger), which helps with memory-bound workload like these wave simulations, providing the memory access pattern is cache-friendly. Architecture-specific fine-tuning, such as the number of threads per thread block, the size of shared memory allocated, and the register allocation per thread, may improve the performance of A100.

5.6 Multi-GPU Implementation and Optimization

In Section 4.1, it has been shown that `p4est` is a powerful library for generating, partitioning, and handling mesh across multiple CPUs that span multiple compute nodes. Unfortunately, at the time of writing, `p4est` does not have built-in support for GPUs. Instead of writing an adaptive mesh refinement library for GPU

from scratch, the strategy for adapting `p4est` to support multi-GPU simulation is outlined in this section. First, the mesh is generated, partitioned, and distributed across MPI processes running on a dedicated CPU core using `p4est`. Then, each MPI process copies the mesh data and mesh structure to GPU memory following the modification to the data structure discussed in Section 5.1.

Once the required data is on GPU memory, the simulation can be performed just like using single GPU, as discussed in Sections 5.3 and 5.4, except there are ghost exchanges between GPUs, which will be addressed in Section 5.6.1. It is imperative to assign one MPI process to handle one GPU to achieve this strategy. Therefore, the number of MPI ranks on a compute node equals the number of GPUs on that compute node. This means a compute node with eight GPUs will have eight MPI processes. Local MPI rank identification is used to choose which GPU index it is assigned to.

5.6.1 Ghost Layer and Ghost Buffer Implementation

Section 4.5 has discussed the ghost exchange for implementing the multi-CPU support for the wave simulations. In summary, the ghost exchange relies on two essential data structures: the ghost layer and the ghost buffer. The ghost layer is an array that stores the neighboring elements' data on other CPUs (i.e., ghost elements), as shown in Figure 4.11. The MPI process receives and stores the ghost elements' data from other MPI processes on this array. During flux computation, the quadrant ID is the index into the ghost layer whenever the neighboring element is a ghost element. The LUT-based neighbor search on GPU (Figure 5.1) adopts this principle by storing index to ghost layer for neighboring elements where property value equal to 2 (i.e., ghost element). On the other hand, the ghost buffer is an array used for staging the elements' data that become the ghost elements for other MPI processes before sending them during the ghost exchange process, as shown in Figure 4.12.

To perform ghost exchange directly on GPU by launching the ghost exchange

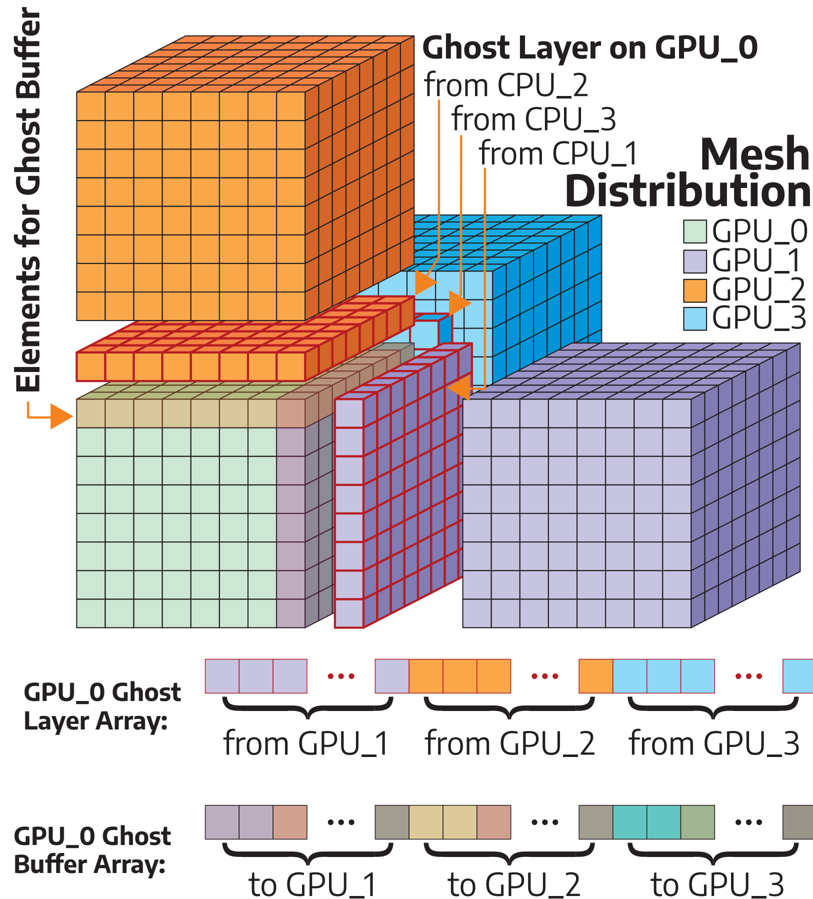


Figure 5.13: A simplified illustration of how the ghost layer array and ghost buffer array are implemented in GPU. It follows the same principle illustrated in Figures 4.11 and 4.12 for the ghost layer and ghost buffer in CPU, respectively. However, the ghost layer and ghost buffer are stored in GPU memory, allowing the ghost exchange kernel to be invoked directly on GPU. While the ghost layer data is automatically updated when receiving the ghost elements' data from other MPI processes, the ghost buffer is not; a special GPU kernel is developed to update the ghost buffer before sending it to the appropriate MPI processes.

kernel, the ghost layer and ghost buffer must be stored inside the GPU memory. This eliminates the need to copy the data between CPU and GPU memory for ghost exchange. Using CUDA-aware MPI, discussed in Sections 2.4.2.4 and 5.6.3, the ghost layer data is automatically updated once the MPI process receives the ghost elements' data from other MPI processes. The data arrangement inside the ghost layer follows the `p4est` indexing scheme, as shown in Figure 5.13. On the other hand, the ghost buffer must be updated manually since it is no longer automatically updated by `p4est` as it is inside the GPU memory.

To update the ghost buffer, a GPU kernel is developed to copy the appropriate elements' data that becomes the ghost elements for other MPI processes (i.e., mirror elements) and collect them into the ghost buffer array according to the indexing scheme shown in Figure 5.13. An efficient mapping scheme is developed to quickly determine the index of mirror elements on the array of (local) quadrants (i.e., the array of `ElementDataBase`) to the ghost buffer array. Since a mirror element can become a ghost element for more than one MPI process, it may be copied more than once. Like receiving the ghost layer, the ghost buffer can be sent directly to appropriate MPI processes leveraging CUDA-Aware MPI.

5.6.2 Ghost Exchange Optimization

The ghost exchange in the CPU code discussed in Section 4.5 uses the same data structure for standard (local) and ghost elements. The `p4est` does not allow using different data structures for ghost elements since it will affect how the mirror elements are copied to the ghost buffer during the ghost exchange. This is inefficient since not all data from the elements need to be exchanged during the ghost exchange.

A dedicated ghost data structure, called `ElementDataBaseGhostV1`, is created to reduce the volume of data during the ghost exchange, containing only `variables` and `materials`. These two members of `ElementDataBase` are the only members[21]

[21] Refer to Table 4.1 and Appendix B.1 for the list of members inside the `ElementDataBase`.

Problem	Precision	# Nodes per Element	Size (Bytes)	
			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

Table 5.2: Standard and Ghost Element Size for 512-node Elements

needed to transfer during ghost exchanges. This reduces the size of the ghost element significantly; its size is only 27% of the size of the standard element, as shown in Table 5.2. This translates to an average of 72.81% and 75.83% lower communication overhead for single-node multi-GPU and multi-node multi-GPU runs, respectively, as shown in Figure 5.14. Further ghost element size reduction will be discussed in Section 6.3, introducing the GhostV2 called `ElementDataBaseGhostV2`.

5.6.3 CUDA-aware MPI with Asynchronous Progression Support

A message-passing library is vital in multi-node multi-GPU simulation as it determines overall simulation performance, mainly when communication between GPUs must occur through the internode communication link, which is often the weakest link in the HPC cluster. Section 2.4.2 has briefly discussed the Message-Passing Interface (MPI). While many different MPI implementations are available, two essential features are CUDA Awareness and Asynchronous Progression, as given in Table 5.3.

MPI Name	Reference	Developer	CUDA-Aware	Async. Progress
MVAPICH[1]	Panda et al. (2021)	Ohio State University.	Yes	Yes
MPICH	Gropp et al. (1996)	ANL	Yes	Yes
OpenMPI[2]	Gabriel et al. (2004)	Univ. of Tennessee, LANL, Indiana University	Yes	No
Spectrum MPI	IBM Corporation (2024)	IBM Corporation	Yes	Yes
Intel MPI	Intel Corporation (2024)	Intel Corporation	No	Yes

Table 5.3: Feature Comparison of MPI Library Implementations. The MVAPICH2-GDR used is version 2.3.4, while the OpenMPI used is version 4.1.1, built from scratch with UCX 1.11.2 and `gdrcopy` 2.3.

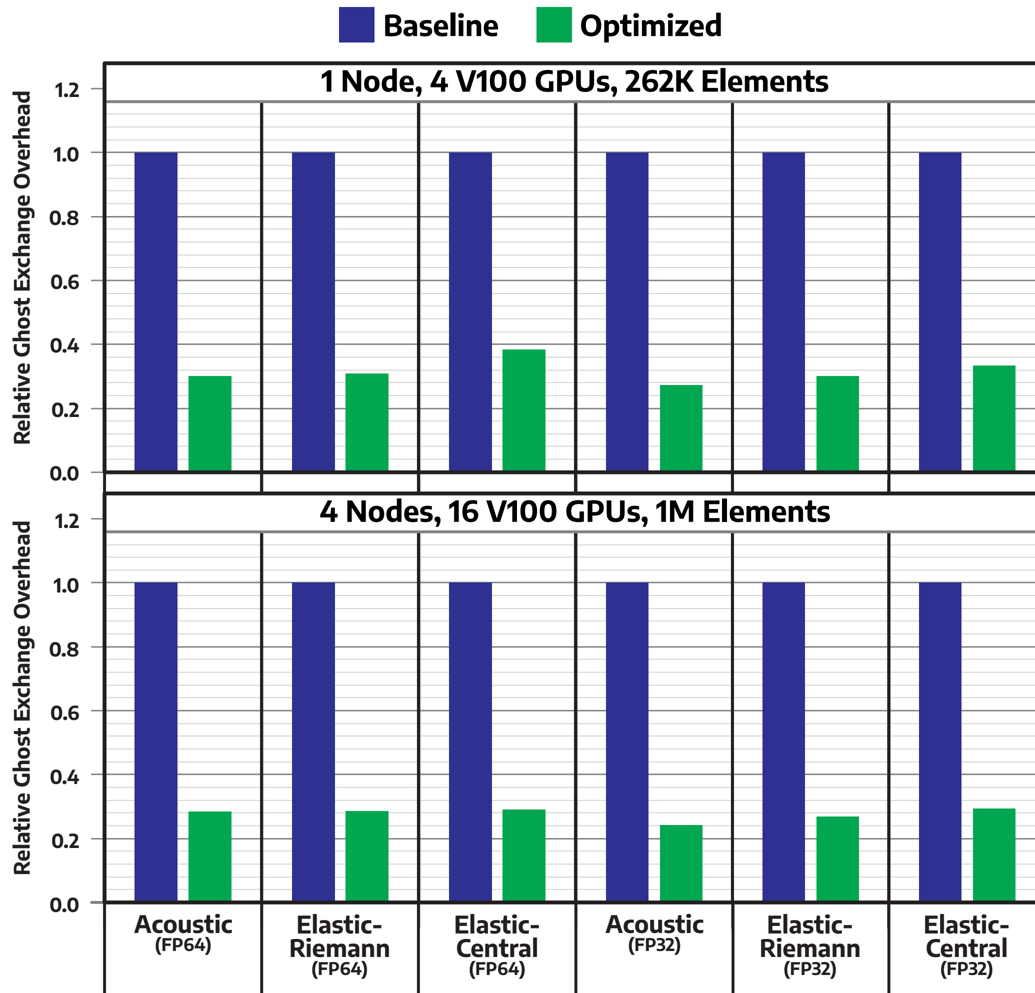


Figure 5.14: The improvement of multi-GPU simulation performance after optimizing the ghost exchange by reducing the size of ghost elements being exchanged as shown in Table 5.2. On average, the simulation time is reduced by 73% where multi-node multi-GPU runs benefit most.

As discussed in Section 2.4.2.4, MPI with CUDA-aware can handle buffers stored inside GPU memory from which the data is sent or to which the data is received. Then, leveraging NVIDIA GPUDirect technology, the data can be exchanged between GPUs with low communication overhead and latency, both intra- and internode. NVIDIA GPUDirect consists of GPUDirect P2P and GPUDirect RDMA. The former allows exchanging buffers inside GPU memory between GPUs inside the same compute node using the fastest communication link (e.g., NVLink). On the other hand, the latter allows sending and receiving data from or to buffers stored in GPU memory through a network adapter (e.g., InfiniBand network interface card) without staging the data first on CPU memory, as described by Li et al. (2020a, 2018a). In other words, with GPUDirect RDMA, the data does not need to be copied from GPU memory to CPU memory before sending it through other MPI processes. Likewise, when receiving the data from other MPI processes, the received data is directly stored inside the GPU memory. This is extremely important since eliminating data movement between CPU and GPU can significantly reduce the communication overhead.

The second crucial feature, asynchronous progression, allows data exchange in the background to overlap communication with computation. However, the MPI standard does not guarantee asynchronous progression even when non-blocking send (`MPI_Isend`) and receive (`MPI_Irecv`) are being used, as described by Message Passing Interface Forum (2015); Laguna et al. (2019). The actual data transfer is usually delayed until `MPI_Wait` is issued, reducing the ability to overlap communication and computation. Some MPI implementations, shown in Table 5.3, support asynchronous progression if explicitly enabled, as described by Horikoshi et al. (2022). This feature should allow hiding communication overhead during ghost exchange by overlapping it with volume and internal flux computation (Figure 5.4), improving overall performance in multi-node multi-GPU runs.

Figure 5.15 shows the comparison of simulation runtime using different MPI libraries shown in Table 5.3 for single-node and multi-node runs. The experiment was done in the TACC Longhorn cluster with compute node topology shown in Figure 3.2

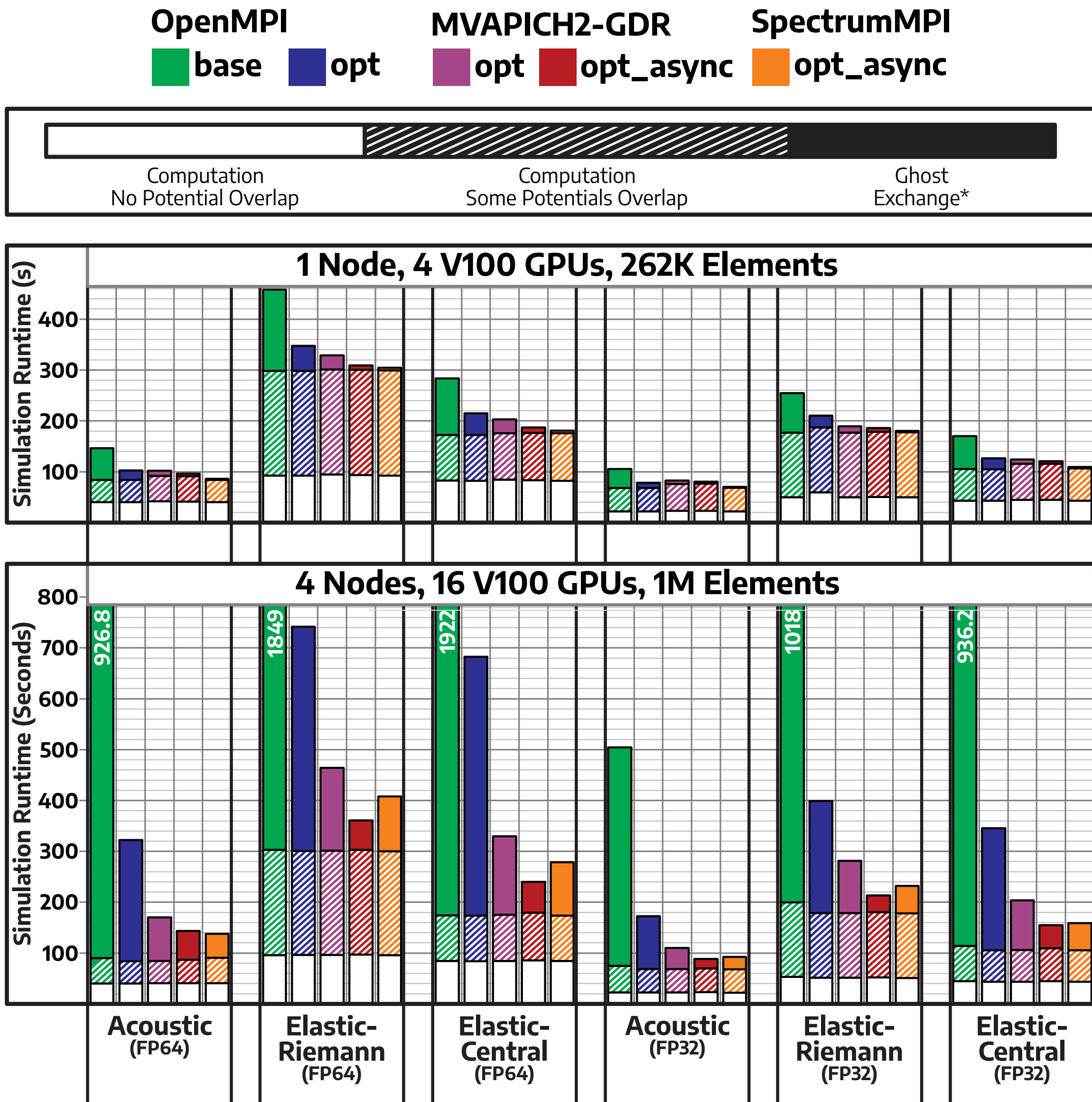


Figure 5.15: The communication overhead (i.e., ghost exchange) in multi-node and multi-GPU runs with different optimization and MPI libraries. The solid color represents the ghost exchange overhead. In contrast, the stripe and white colors indicate the time consumed by computation that can be overlapped (i.e., volume and internal flux) and cannot be overlapped (i.e., external flux and integration), respectively.

and (approximated) cluster network topology shown in Figure 3.3. The base implementation with OpenMPI uses a standard data structure (i.e., `ElementDataBase`) for the ghost element. In contrast, the other implementations use the optimized structure for the ghost element (Table 5.2). Without asynchronous progression explicitly enabled, the MVAPICH2-GDR yields an average of 44.31% and 61.67% communication overhead reduction compared to the optimized version with OpenMPI. Enabling the asynchronous progression for MVAPICH2-GDR further reduces communication overhead by an average of 56.74% compared to MVAPICH2-GDR without it.

For obvious reasons (i.e., IBM-provided software on the IBM platform), the Spectrum MPI with asynchronous progression enabled has an average of 51.27% lower communication overhead on single-node multi-GPU run compared to MVAPICH2-GDR with asynchronous progression enabled. However, it falls behind on multi-node multi-GPU runs with 33% more communication overhead due to the inefficiency of multi-rail InfiniBand utilization (i.e., each compute node has two Network Interface Cards, each connected to one CPU). Nevertheless, asynchronous progress allows for the overlap of ghost exchange with volume and internal flux computation better, thus hiding some communication overhead.

5.6.4 Scalability Evaluation

With the multi-GPU implementation discussed in Section 5.6.1, optimization discussed in Section 5.6.2, and choosing the right MPI library as discussed in Section 5.6.3, the multi-node multi-GPU implementation achieves near-perfect weak-scaling over 128 GPUs on 32 compute nodes. The term weak-scaling implies that, as more GPUs are used, the problem size is also increased, measured as the number of elements (i.e., on the right vertical axis). This weak scaling is shown in Figure 5.16 where SpectrumMPI is used as the MPI library for running elastic wave simulations with the Central flux solver. Other problems (i.e., acoustic, elastic with Riemann flux solver) have the same results. The single-node (four GPUs) run enjoys the benefit of high-bandwidth NVLink as the inter-GPU communication link. On the other hand,

the performance of multi-node multi-GPU is limited by the inter-node communication link (i.e., InfiniBand), especially for FP64 runs.

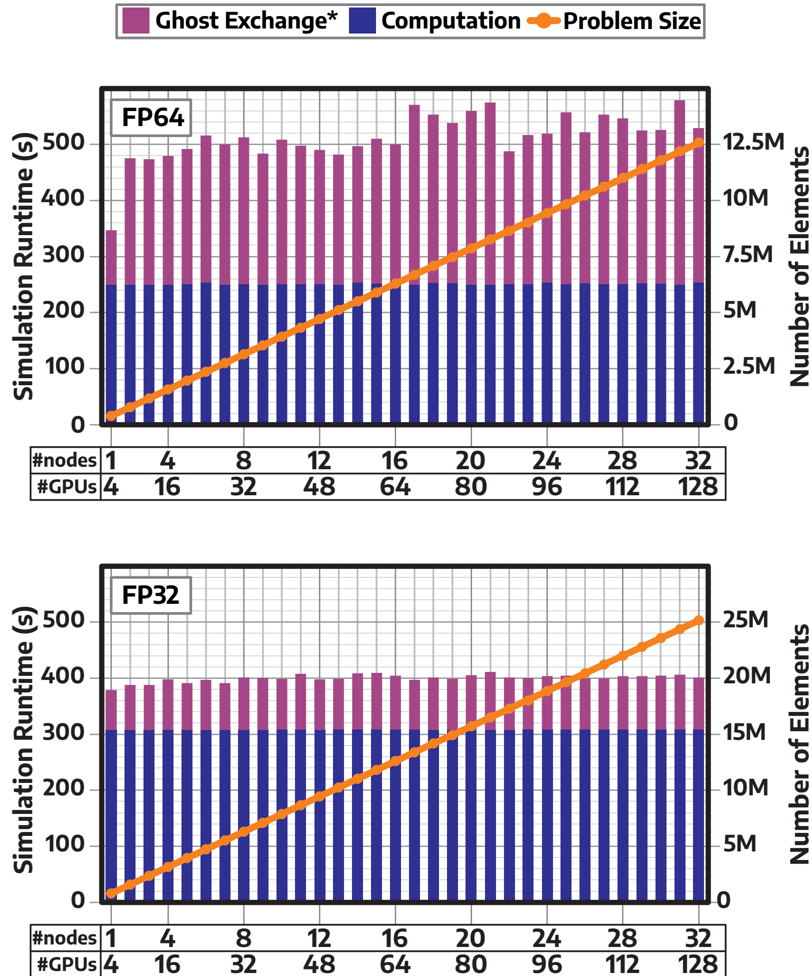


Figure 5.16: The weak scaling achieved by the GPU codes for running elastic wave simulation with central flux solver for FP64 (left) and FP32 (right) using up to 32 nodes (128 GPUs). Note that as more GPUs are added, the problem size, measured by the number of elements, is increased.

Chapter 6: Reducing Communication for dG-based Wave Simulations

This chapter describes the second contribution of this dissertation: developing communication-reducing algorithms to reduce the communication overhead in dG-based wave simulations, based on the key takeaways from Chapter 5. The intra-device communication, in the form of data movement between on-chip and off-chip memory, is found to be the critical bottleneck that limits the attained performance of the GPU, as discussed in Section 5.5.3. Even after applying optimization techniques discussed in Section 5.4 and using GPUs with 900 GBps HBM2 bandwidth, the wave simulations are still memory-bound. To make matters worse, large-scale wave simulations utilizing hundreds of GPUs are also limited by the available bandwidth to perform inter-device communications, both intra-node and inter-node. The latter, being the weakest link in the computing clusters, is shown to be the key bottleneck in Section 5.6.4.

The first problem discussed in this chapter is reducing the intra-device communication overhead: the data movement between the GPU's on-chip and off-chip memory. As mentioned in Section 2.3.1, the reason for using the dG discretization method is the locality of each element, where the operations on each node inside an element are local to the element, which should yield better locality for the hardware. However, due to the hardware constraints, the locality of the element may not be achieved, leading to additional data exchanges between the hardware units. The node-tiling strategy is introduced in Section 6.1 to alleviate this issue. Considering the hardware architecture, the tile size can be adjusted to maintain the locality of elements in each hardware unit, reducing intra-device communication overhead.

Additionally, since the kernel fusion discussed in Section 5.4.1 has been proven to reduce the data movement and kernel launch overhead successfully, a technique

called unified kernel is introduced in Section 6.2 to maximize the benefit of preserving GPU execution states by merging multiple kernels. Instead of only fusing volume and flux kernels, the unified kernel merges volume, flux, and integration kernels. An innovative algorithm technique is developed to overcome the data hazard that becomes the ceiling of the previous implementation. This strategy must also consider multi-GPU implementation, where some nodes of an element need to wait for ghost exchange to finish before they complete the external flux computation and the integration can be performed.

In addition, this chapter also discusses a communication-reducing algorithm for inter-device communication. Limited bandwidth in inter-device communication interface, especially for inter-node communication, is the leading performance bottleneck, as evident in Section Section 5.6. With the success of reducing the volume of data during the ghost exchange, as discussed in Section 5.6.2, additional efforts are made in this chapter to reduce the data volume further. Face-Node-Only ghost exchange, discussed in Section 6.3, modifies the data structure of an element only to include the `variables` of the nodes located on the faces of the element (i.e., external-facing nodes). However, this strategy may require additional overhead to calculate the node index mapping.

Furthermore, adopting the technique of mixed-precision computing from the machine-learning crowd, a reduced-precision ghost exchange is introduced in Section 6.4. By using smaller data types, such as single-precision and half-precision, the volume of data can be halved and quartered, respectively, compared to double-precision. However, the reduction of precision will impact the numerical accuracy, and thus, the feasibility of this strategy must be evaluated, including whether it will cause numerical instability. Last but not least, an exciting technique called partial ghost exchange is discussed in Section 6.5. Unlike standard ghost exchange, where all `variables` of the nodes are exchanged, only several of them are sent, and hence, partial ghost exchange. The rest are approximated by performing computation on the receiving side. This will significantly reduce the volume of data at the expense of

additional computation. Although this idea is still in the early exploration stage, it is worth mentioning in this dissertation.

Finally, the performance evaluation of the communication-reducing algorithms developed in this chapter is performed. Sections 6.6 and 6.7 provide detailed explanation on evaluating the intra-device and inter-device communication-reducing algorithms, respectively. In summary, the algorithms presented in this chapter provide excellent improvements to the performance of dG-based wave simulations thanks to the lower intra-device and inter-device communication overheads.

6.1 Achieving dG Element Locality with Node Tiling

This section outlines the strategies to harness the element locality from dG discretization (Section 2.3.1), where the computation inside an element (i.e., interior computation) is always local to the element without needing data exchange from neighboring elements. This property of dG must be exploited since it can reduce intra-device data movement when executing kernel on the interior of elements, such as volume and integration kernels. This is the ideal execution, discussed in Section 6.1.1, where the data of an element is always kept near the ALUs throughout the kernel execution on that element. However, exploiting element locality is not always straightforward due to the target hardware's limited resources (i.e., registers, first-level cache), as discussed in Section 6.1.2. An algorithmic technique called Node-Tiling is developed to achieve element locality of dG, reducing the intra-device communication between the memory hierarchies, as discussed in Section 6.1.3.

6.1.1 Ideal Execution for Harnessing dG Locality

One of the advantages of using dG discretization is locality; the interior computations, such as volume and integration, are local to each element. Theoretically, once the whole element is loaded into the on-chip memory near the ALUs (e.g., registers or first-level cache), the calculation can be performed quickly with minimal data

transfers from and to the higher-level memory hierarchies. In the case of volume computation, each node within an element can be processed in parallel since they are not dependent on each other; no intermediate results from one node become the operands for different nodes. However, each node needs `variables` data from other nodes when calculating gradient p and divergent $\mathbf{v}$ for acoustic wave simulations, and divergent of $\mathbf{S}$ and derivative of $\mathbf{v}$ for elastic wave simulations (Figure B.9). This should not be a problem if data of all nodes within an element is stored nearby at lower-level on-chip memory without the need to fetch the data from a higher-level memory. In the case of integration, the computation is simpler since each node only needs its own `contributions`, `mass_inverse`, and `auxiliary`, without taking any data from other nodes.

6.1.1.1 Mapping Elements into Hardware

Looking at the GPU hardware and software perspective, discussed in Section 2.6.2 and illustrated in Figure 2.6, one thread block is scheduled into one Streaming Multiprocessor (SM). A thread block contains up to 1024 threads, organized in groups called warps, each containing 32 threads. The availability of hardware resources in an SM, which includes the registers, the L1 cache, and the shared memory, impacts how a thread block is executed. The ideal execution strategy is to map one element into one thread block. This way, once the thread block is scheduled for execution into an SM, the whole element data will be brought into the on-chip memory of the SM (i.e., L1 cache and registers).

Assuming the SM has enough registers and L1 cache, the whole element data remains there throughout the execution of the thread block. This reduces intra-device communication since minimal data will be exchanged between on-chip memory inside the SM, L2 cache, and off-chip HBM2 memory while executing one thread block. In addition, to extract the node-level parallelism, the number of threads inside the thread block should equal the number of nodes in each element. In this case, a 512-node element will be handled by one thread block containing 512 threads, dedicating

one thread per node. This means 16 warps will be scheduled for execution to four SM subpartitions (SMSPs) inside the SM. Since all SMSPs share the same L1 cache, the element data can be fetched quickly.

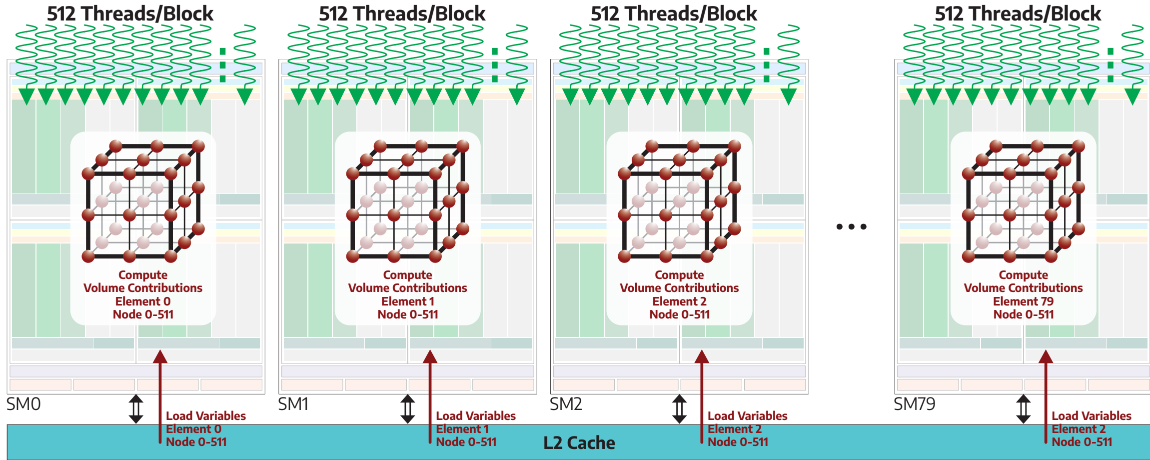


Figure 6.1: The ideal execution for a thread-block handling one element. Each thread block is mapped into one SM and has 512 threads corresponding to the 512 nodes inside the element, allowing for maximizing node-level parallelism. Since, ideally, the whole element fits into the on-chip memory inside an SM, only minimal data traffic from and to the higher-level memory hierarchies once the element's data is loaded.

6.1.1.2 Streaming Multiprocessor as Element Processor

Figure 6.1 shows what the ideal execution looks like. The SM in GPU is now essentially becoming an *Element Processor* dedicated to processing one element at a time. The term *Element Processor* has been used by previous work by Gourounas et al. (2023b,a) to call a dedicated structure designed to accommodate the fast processing of an element with ample on-chip memory to store the data.

Since GPU has multiple SMs, the element-level parallelism can also be extracted since the SMs can process individual elements simultaneously without data exchanges between SMs. However, the number of elements in a mesh far exceeds the number of SMs available in GPU. The SMs process a subset of the mesh before moving to another subset. It is called element-tiling, and as long as one element is

handled by one thread block, it will not be split across multiple SM. Note that the programmer cannot specify the sequence in which the GPU schedules each thread block (i.e., processes each element) to an SM; it is up to the scheduler engine inside the GPU.

6.1.2 Locality of Element and Hardware Limit

On paper, the ideal execution, discussed earlier in Section 6.1.1 allows for harnessing the locality resulting from the use of dG method for discretizing the problem domain, which is essential since the interior element computations (i.e., volume kernel and integration kernel) benefit most from the local, fast-access to element data stored inside on-chip memory in the SM. However, when it is implemented on the hardware, several challenges must be considered, as follows.

6.1.2.1 Potential Eviction of Element's Data to Off-chip Memory

The ideal execution works beautifully if the hardware resources in each element can support having so many threads and storing the whole element's data. According to Table 4.1, the largest size of an element is below 128 KB (i.e., elastic wave simulations with double precision arithmetic). Based on the data from NVIDIA Corporation (2017), L1 cache capacity per SM in NVIDIA Tesla V100 is 128 KB, sufficient for storing the whole element, providing that no shared memory is used.

On the other hand, the register capacity per SM is 256 KB. With 512 threads per thread block, each thread has a register allocation of 512 bytes, enough for storing 64 double-precision values or 128 single-precision values. The per-thread register allocation may be insufficient, especially when dealing with the computation with many intermediate results. Since it has more intermediate results, the effect of per-thread limited register allocation becomes more apparent for long kernels, such as the fused volume and internal flux. Some data will spill over from registers to the local memory, private memory space for each thread (Section 2.6.3.1). On hardware, the

private memory is stored in the L1 cache; if insufficient, it will be stored in the L2 cache and the off-chip HBM2 memory.

This means there is competition for using the L1 cache: storing the element's data and storing the data from register spillage. While the hardware will automatically manage the L1 cache based on the replacement policy, there is a possibility that the element's data will be evicted from the L1 cache of an SM to the L2 cache. Using shared memory, discussed in Section 5.4.2, forces the crucial data of the element (i.e., `contributions`, `variables`) to be kept inside the on-chip memory of the SM, eliminating eviction.

Although the element's data can potentially be evicted to the L2 cache, which is still inside the chip, the capacity of the L2 cache is limited. With a total L2 cache capacity of 6144 KB shared across 80 SMs, the average L2 cache per SM is only 76.8 KB, less than the capacity of the L1 cache per SM. Thus, there is a possibility that the element's data will be evicted from the L2 cache and must be fetched again from the off-chip HBM2 memory. This creates much intra-device communication traffic between the on-chip and off-chip memory, limiting the attained performance, as discussed in Section 5.5.3.

6.1.2.2 Scheduling Multiple Thread Blocks to Single SM

Previously, only one thread block is considered per SM, allocating all of the hardware resources inside an SM to only one thread block. However, as discussed in Section 2.6.2.3, scheduling only one thread block into one SM is not ideal since it limits the ability of GPU to hide memory access latency through aggressive context switching. Multiple thread blocks should be scheduled and co-located into the same SM, allowing the scheduler to context-switch to another thread block when the execution of one thread block is stalled due to memory access. The SM Occupancy metric describes how many warps across the thread blocks can occupy an SM.

The optimization strategy discussed in Section 5.4.3 tries to increase SM occu-

pancy by carefully allocating registers per thread and limiting the number of threads per thread block. Instead of using 512 threads per thread block corresponding to the number of nodes inside 512-node elements, it reduces the number of threads to 128 or 256, depending on the kernels. For example, in Elastic Riemann, the number of threads per thread block should be reduced to 128; in Acoustic Riemann, the number of threads per thread block should be reduced to 256. The former has more intermediate results, requiring more registers per thread. Either way, it allows scheduling more than one thread block to an SM, increasing the SM occupancy and allowing the GPU to hide memory access latency better. Lower-order elements, such as the 64-node element discussed in Section 5.5.4, will have better SM occupancy since they require smaller thread blocks (i.e., 64 threads per thread block). However, the simulation performance using mesh with smaller order elements is limited by the significant amount of non-local computations (i.e., flux) compared to mesh with higher order elements, as shown in Section 5.5.4.

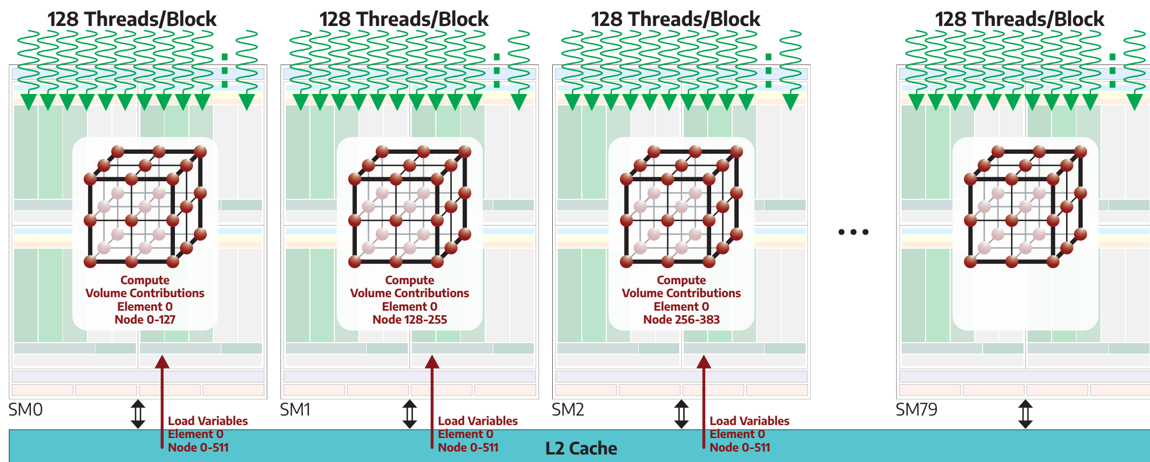


Figure 6.2: The SM-Occupancy-Aware execution for a thread-block handling one element. To allow for scheduling multiple thread blocks inside a single SM, hardware resource usage for each thread block must be limited. This includes reducing the number of threads per thread block and limiting the number of registers per thread, which reduces the number of registers required per thread block.

Figure 6.2 shows the SM-Occupancy-Aware execution, which has been imple-

mented with optimization described in Section 5.4.3. With this execution strategy, the thread block size is smaller, only containing 128 or 256 threads. However, since the number of threads per thread block is less than the number of nodes per element, the element is processed by multiple thread blocks, which are potentially scheduled into different SMs. Programmers have no control over how the thread blocks are scheduled. A thread block working on a part of the element may be scheduled into an SM later, while the other thread blocks handling the other parts of the same element finish their execution on different SMs. This situation makes the interior computation of an element no longer local from a hardware perspective; there is some communication traffic between SMs to perform the interior calculation of the single element. With many SMs working on different elements, this traffic can add up, limiting the simulation performance.

In addition, in the volume kernel, although each thread block only needs to work on the part of the elements, it still needs to fetch all of the element's data for all nodes, except for `contributions`. If all thread blocks are executed simultaneously on different SMs, it increases pressure for the L2 cache since it must bring the same data to multiple SMs. These data are duplicated into multiple SMs (i.e., L1 cache inside the SM), which is an inefficient use of the L1 cache across all SMs. Instead of storing multiple different data that is frequently used, it holds duplicated data. On the other hand, if not all thread blocks are executed simultaneously, the latter thread blocks will most likely need to access the element's data from the off-chip HBM2 memory since it may have been evicted from the L2 cache due to its limited capacity. This, again, increases data movement between on-chip and off-chip memory, whose impact may be partially hidden by the higher SM Occupancy.

6.1.2.3 Shared Memory Usage Consideration

Finally, as discussed in Section 5.4.2, using this strategy, only `contributions` is worth storing inside the shared memory. When running the volume and flux kernels on 128 out of 512 element nodes (i.e., 128 threads per thread block), the kernel only

needs `contributions` corresponding to the nodes. However, the kernels still need the `variables` from all nodes inside the element. Aside from its complex memory access pattern, storing `variables` in shared memory is not worth it since it consumes a lot of shared memory spaces, reducing the L1 capacity without any meaningful performance improvements.

6.1.3 Modifying Execution Strategy using Node Tiling

As described in Section 6.1.1, ideal execution provides the best locality for processing each element. However, this execution strategy is not always possible and produces positive performance improvements, as hardware resources limit it. Meanwhile, the SM-Occupancy-Aware execution allows the GPU to perform its latency-hiding better through aggressive context switching. However, this strategy makes local computation of an element no longer local as it splits across multiple SMs, generating more intra-device communication traffic. This section describes an execution strategy that combines the best of ideal execution with the best of SM-Occupancy-Aware. This strategy is called Node-Tiling Execution.

6.1.3.1 Dividing Element into Multiple Tiles

Figure 6.3 shows what the node-tiling execution strategy looks like. Like in SM-Occupancy-Aware execution, the number of threads in each thread block is reduced. In a mesh with 512-node elements, the thread block size can be 128 and 256, $\frac{1}{4}$ and $\frac{1}{2}$ the number of nodes inside each element, respectively. However, instead of splitting the elements into four or eight different thread blocks, each element is handled only by one thread block, like in ideal execution. The threads inside the thread block work on a subset of nodes, which then continue their executions for another subset of nodes.

This subset of nodes is called a tile, and the number of threads inside the thread block depends on the tile size. For example, if the tile size is 128, the number

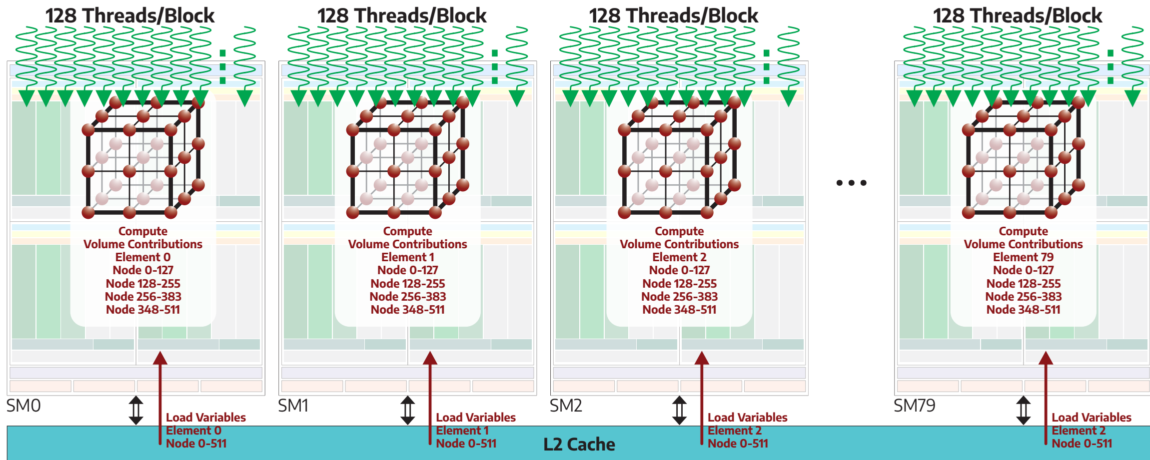


Figure 6.3: The node-tiling execution strategy combines the best things in SM-Occupancy-Aware execution while maintaining element locality like in ideal execution. One element is handled exactly by one thread block, but the number of thread blocks is no longer equal to the number of nodes in an element. Instead, the threads work on a subset of nodes, called tile, one at a time.

of threads in a thread block is 128. There are four subsets of nodes for an element with 512 nodes, each with 128 nodes. The thread block will perform computation on nodes 0 to 127, 128 to 255, nodes 256 to 383, and nodes 384 to 511. In other words, instead of having parallelism of $N_{element} \times N_{nodes_per_element}$ for interior element computation, the node tiling execution strategy has parallelism of $N_{element} \times N_{nodes_per_tile}$. The tile size can be adjusted to the availability of the hardware resources in each SM on a particular GPU architecture.

6.1.3.2 Nodes Execution Flow

Execution flows for each node inside an element are shown in Figure 6.4, comparing ideal, SM-Occupany-Aware, and Node-Tiling strategies. The ideal execution uses wider thread blocks whose number of threads equals the number of nodes in an element. The SM-Occupancy-Aware execution uses multiple narrower thread blocks to manage hardware resource utilization but splits an element into multiple SMs. The node-tiling combines the best of both strategies; it uses a narrower thread block to

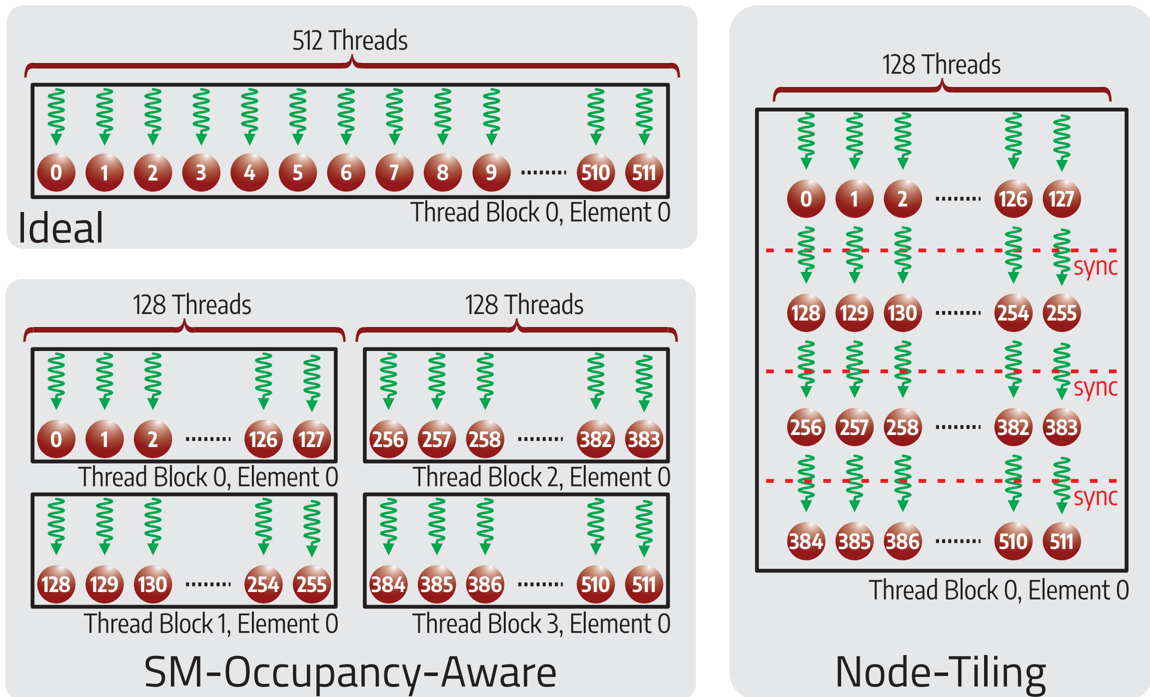


Figure 6.4: The comparison of execution flow for each node inside an element for Ideal, SM-Occupancy-Aware, and Node-Tiling strategy. The ideal execution uses a wider thread block where the number of threads equals the number of nodes. In contrast, SM-Occupancy-Aware execution uses multiple narrower thread blocks, splitting the element into multiple SMs. The Node-tiling combines the best of both Ideal and SM-Occupancy-Aware, using narrower thread blocks while keeping an element inside single SM, maintaining element locality.

help manage hardware resource utilization while at the same time maintaining the locality of an element, harnessing the goodness of the dG discretization better and achieving higher SM occupancy.

After executing a tile, there is a thread-block-wide synchronization barrier (`__syncthreads()`) to ensure all threads finish executing the tile before moving to the next, eliminating any race conditions. If a single thread handles one node throughout the execution, this synchronization barrier should not be needed. A static `for` loop can be used to implement iteration to execute each tile instead of writing the same codes multiple times, making debugging and modifying the code easier. The compiler

will automatically unroll the static `for` loop or instruct it to unroll using `#pragma unroll`.

6.1.3.3 Shared Memory Usage Consideration

Storing both `variables` and `contributions` may yield improved performance by guaranteeing that an element is always processed by one thread block mapped into one SM. Both are important and used repeatedly. Thus, ensuring both stay inside the shared memory of an SM will eliminate the possibility of being evicted to the off-chip memory. However, storing both may not be the best for all cases, depending on the reliance on the L1 cache and the capacity of the L1 cache of a particular GPU architecture.

6.2 Preserving GPU Execution States using Unified Kernel

The optimization technique discussed in Section 5.4.1 significantly improves the wave simulations' performance, as shown in Section 5.5.1. This is achieved by fusing two simulation kernels, the volume and flux, into one kernel, reducing the kernel launch overhead and the intra-device communications. Since the previous exploration was successful, extending this optimization technique is promising. The question remains: how far can this kernel fusion technique be applied to wave simulation applications? Is there any algorithm technique that allows pushing the boundary that limit previous implementations?

This section discusses the intra-device communication reducing algorithm to develop a more advanced kernel fusion technique as an extension to Section 5.4.1. Due to the data hazard and the need for implicit device-wide synchronization, the previous technique could only merge volume and flux kernels. Utilizing innovative algorithms, it is now possible to merge all three simulation kernels, volume, flux, and integration, into one unified kernel. First, a modification to the simulation flow is performed by considering multi-GPU support, as discussed in Section 6.2.1. Then, the

data hazard problem that limits previous kernel fusion implementations is discussed briefly in Section 6.2.2 as a reminder to the readers.

Next, the double-buffering technique, a common technique for addressing the data hazard problem, is discussed in Section 6.2.3. This naïve technique has many shortcomings, making it unsuitable for implementing the unified kernel. To overcome these issues, the proposed double buffering technique is discussed in Section 6.2.4. It is designed to efficiently implement double buffering without the additional buffers, without the need for device-wide synchronization barriers, and with minimal modification to the existing codes. In addition, it is best to combine with shared memory usage discussed in Sections 5.4.2 and 6.1.2.3. Finally, after the ingredients are ready, the implementation of the unified kernel is discussed in Section 6.2.5.

6.2.1 Modifying The Simulation Flow

The first step of implementing the unified kernel is to modify the simulation flow. Previous kernel fusion technique already modified the simulation flow on CPU, illustrated in Figure 4.9, into the simulation flow on GPU using fused volume and flux kernel, illustrated in Figure 5.4. This involves separating the flux kernel into two parts, internal and external, to consider the ghost exchanges required for multi-GPU implementation, discussed in Section 5.6. Then, the fusion is performed by merging volume and internal flux kernels.

The modification for implementing the unified kernel is shown in Figure 6.5. It uses the same approach of separating internal flux from external flux. However, the integration is no longer a separate kernel and is no longer launched at the end of the integration step. Instead, it is merged with volume and internal flux kernels. This way, whenever a node inside an element finishes its volume and flux `contributions` computation, it can directly proceed to compute the integration and update the `variables` for the next time step without needing a separate kernel launch.

However, in the multi-GPU scenario, there is a ghost exchange that needs to

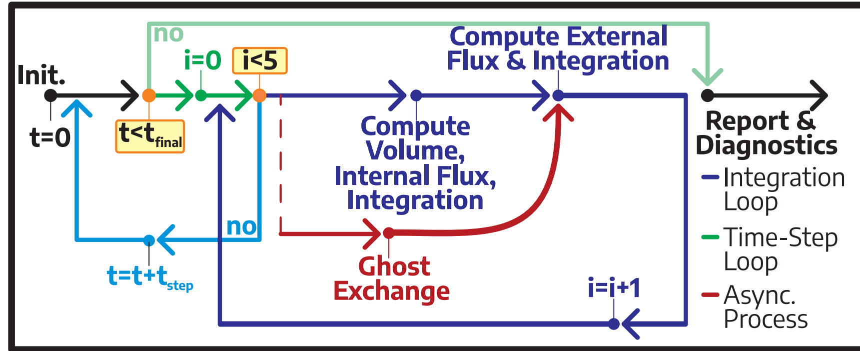


Figure 6.5: The modification to the simulation flow by fusing the volume, flux, and integration as an effort of optimization to reduce intra-device communication and kernel launch overhead. Following the strategy in the kernel fusion (Section 5.4.1), the flux kernel is separated into two parts, internal and external, to facilitate multi-GPU implementation (Section 5.6).

be taken into account. In this case, a node may be part of element faces, where the neighbor is a ghost and needs to wait for the ghost exchange to finish before completing the flux computation. To accommodate this issue, the node can skip the integration, wait for the ghost exchange to finish, and perform external flux computation using the updated ghost elements. Once the external flux computation is done, the integration can be performed directly for that particular node without a separate kernel launch.

With this simulation flow modification, only two main kernels are present: the fused volume, internal flux, and integration kernel, as well as the fused external flux and integration kernel. The fused volume, internal flux, and integration kernel have a longer execution time than the fused volume and internal flux kernel, giving more room for MPI asynchronous progress to better overlap communication with computation during the ghost exchange (Section 5.6.3).

6.2.2 Data Hazard Problem

On paper, the modification of the simulation flow looks straightforward. However, there is a major bottleneck that needs to be taken care of, that is the data

hazard between the use of `variables` to compute `contributions` (i.e., volume and flux) and the update of `variables` by the integration kernel. This limits the previous kernel fusion implementation, allowing only volume and flux kernels to be fused. Section 6.2.2 illustrates the data hazard problem discussed in this section. There are two data hazards: intra-element data hazard and inter-element data hazard.

6.2.2.1 Intra-Element Data Hazard

The intra-element data hazard appears between the nodes inside the same element. While the nodes within an element can compute the `contributions` in parallel without depending on intermediate results from other nodes, they need to access the `variables` from other nodes. For example, during the volume computation, specifically when calculating gradient p and divergent $\mathbf{v}$ for acoustic wave simulations, and divergent of $\mathbf{S}$ and derivative of $\mathbf{v}$ for elastic wave simulations (Figure B.9), the node needs data from other nodes located at the same x-axis, y-axis, and z-axis.

The thread block is divided into several warps, each containing 32 threads (Section 2.6.2). For example, a thread block with 512 threads contains 16 warps. The scheduler distributes the warps into four SM Subpartitions (SMSPs) inside an SM. Since the number of warps are larger than the available SMSPs, they are executed with time-slicing. One warp may have finished its execution, where the 32 nodes processed by 32 threads have updated the `variables` for the next integration loop or time step after performing the integration. When another warp starts its execution, it will take the newly-updated `variables` instead of the older one, resulting in incorrect computations.

If one thread block can handle one element, just like what Node-tiling in Section 6.1.3 accomplished, then using an explicit thread-block wide synchronization barrier, `__syncthreads()`, can avoid intra-element data hazard. The barrier is placed just before integration, ensuring all threads within the thread block finish computing volume and flux contributions for all nodes before entering the integration. It is non-

dependent on how the warps are scheduled for the SMSPs. The threads in the warps will have to wait at the barrier until other threads in other warps reach the barrier.

On the other hand, if multiple thread blocks are needed to handle one element (i.e., the element is split into multiple thread blocks), just like what SM-Occupancy-Aware execution in Section 6.1.2) achieved, then the solution becomes more complicated. Since the programmers cannot control how the thread blocks are scheduled into the SMs, an explicit device-wide (kernel-grid-wide) synchronization barrier must be used to avoid the intra-element data hazard, which is now split into multiple thread blocks. However, this type of synchronization barrier is expensive and not supported by most GPUs. An implicit barrier involving performing operations on different kernels is usually used for device-wide synchronization. This approach is used by kernel fusion discussed in Section 5.4.1, keeping the integration separate. Another approach is using the same solution as inter-element data hazard (i.e., double buffering), which will be discussed in Section 6.2.3.

6.2.2.2 Inter-Element Data Hazard

Inter-element data hazard occurs when an element requires `variables` data from its neighbor to perform flux computation. However, the neighbor has already finished the integration computation and updated the `variables` for the next integration loop or time step, leading to incorrect results. Since different thread blocks handle each element and the execution scheduling of the thread blocks is determined by the GPU hardware, an explicit device-wide synchronization barrier is needed to alleviate inter-element data hazards. As described earlier, it is expensive, and most GPU hardware does not support it.

Figure 6.6 illustrates how the scheduling of the execution of thread blocks contributes to inter-element data hazard. The data flow inside each element follows Figure 4.10, albeit drawn with simplification. Consider two elements, Element 0 and Element 1, each processed by different thread blocks. Entering Integration Loop 0,

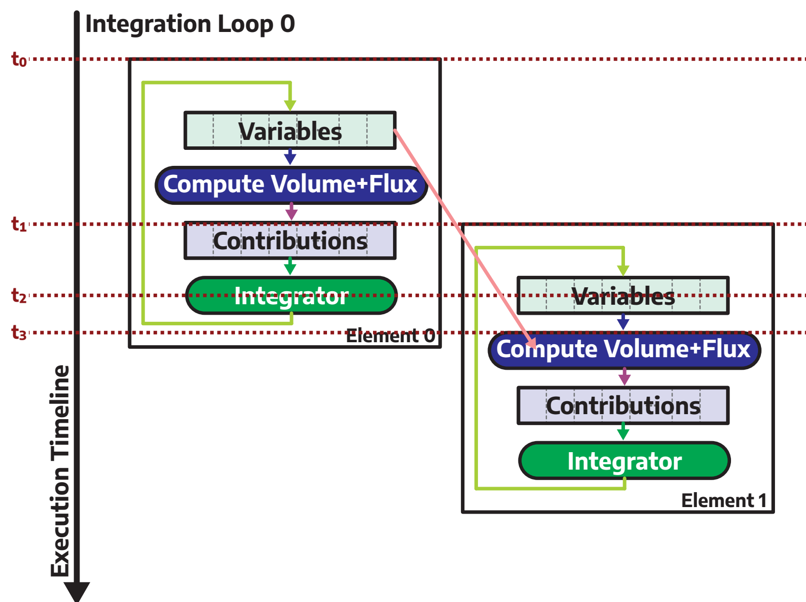


Figure 6.6: The inter-element data hazard occurs when fusing integration kernel with volume and flux kernels. Due to the nature of thread block scheduling, the programmers cannot determine the sequence in which the elements are processed. Two elements may not be processed simultaneously: t_0 for Element 0 and t_1 for Element 1. Once Element 0 updated its `variables` at t_2, Element 1 can no longer obtain the correct `variables` from Element 0 at t_3, leading to incorrect computation.

the thread block handling Element 0 is scheduled to an SM and starts its execution at t_0. A few moments later, at t_1, the thread block handling Element 1 is scheduled to another SM and starts its execution. At t_2, Element 0 updates the `variables` for the next integration loop or time step by running the Integration. However, at t_3, Element 1 starts computing volume and flux contributions. Since Element 0 is a neighbor for Element 1, it obtains `variables` data from Element 0. Unfortunately, since it has been updated for the next integration loop or time step, the `variables` on Element 0 is no longer the correct data for the current integration loop or time step.

6.2.3 Naïve Double Buffering

Double buffering avoids data hazards in parallel execution by utilizing multiple buffers to store old and new values. For example, the works by Gourounas et al. (2023a,b) use double buffering by utilizing two block RAMs in FPGA to implement the element processors. This way, the element can store the updated `variables` for the next integration loop or time step in the second buffer while keeping old `variables` in the first buffer. When another element requires the current `variables` from that element, it can obtain them quickly from the first buffer. Before beginning the next integration loop or the next time step, the contents of the second buffer are copied to the first buffer, making it ready to be used for the next iteration.

Section 6.2.3 illustrates the implementation of double buffering to avoid inter-element data hazards. Each element has two arrays for storing the variables: the `variables` and the `variables*`. Element 0 starts its execution in t_0, and when it performs the integration in t_2, it stores the updated variables in `variables*`. The neighboring element, Element 1, starts its execution at t_1 and performs volume and flux contributions computation at t_3. It obtains the variables data from Element 0 from `variables`, which is the correct data. Once all elements finish their execution at t_4 and before starting the next integration loop at t_5, the `variables` is updated by copying the contents of `variables*`.

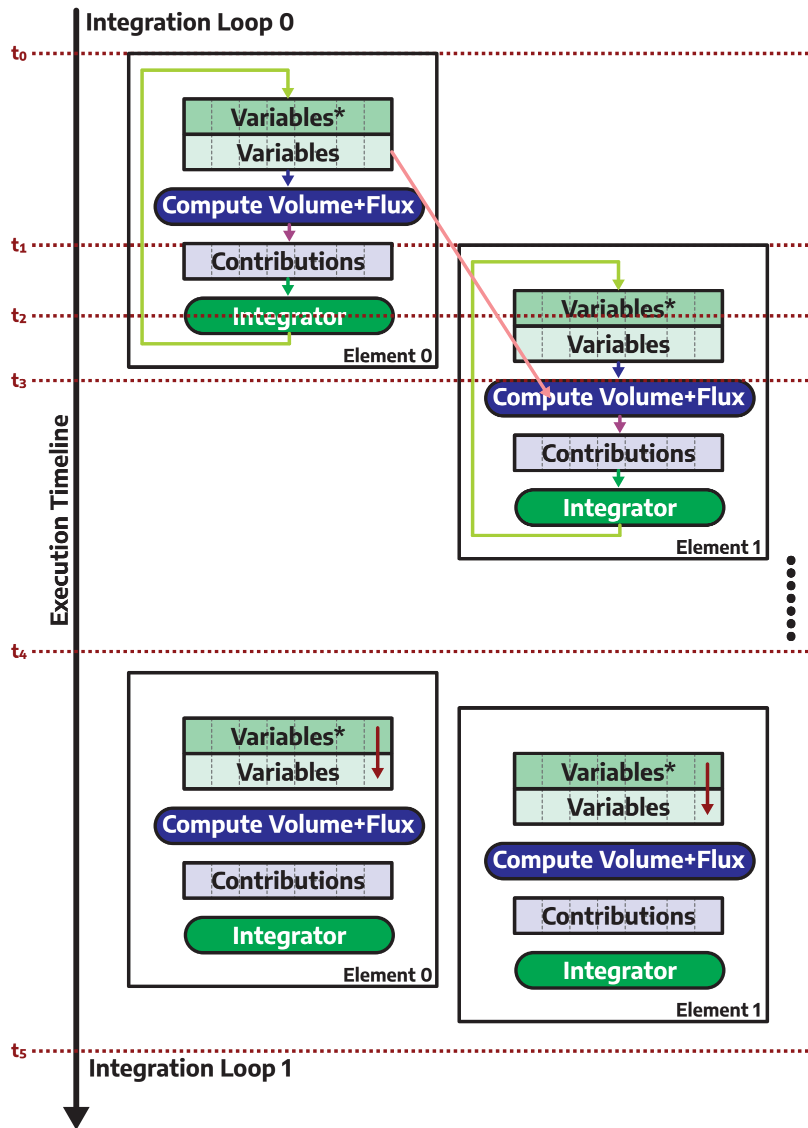


Figure 6.7: The naïve double buffering for avoiding the inter-element data hazards associated with fusing integration kernel with volume and flux kernels. Each element has one additional buffer, `variables*`, to store the updated variables for the next step. When Element 0 starts its execution at t_0 and reaches Integration at t_2, it stores the updated variables for the next integration loop inside the `variables*`. When its neighboring element, Element 1, begins its execution later at t_1 and performs volume and flux computations at t_3, it can correctly obtain the face variables from Element 0 by accessing the `variables`. After all elements completed current integration loop at t_4, all elements copied the contents of `variables*` to `variables`, ready for the next integration loop at t_5.

However, naïve double buffering has three significant drawbacks when implementing a unified kernel for wave simulations. First, it requires additional memory to store the second buffer of `variables`, which increases the size of each element in memory. The memory increase is significant, especially for running wave simulations with thousands of elements. Second, it creates more traffic due to the higher pressure for the L1 cache and registers, introducing more intra-device data movement. Third, all elements must have completed their volume and flux contributions computation to correctly copy the `variables*` into `variables`. This means that there must be a device-wide synchronization barrier at t_4, which is expensive. Alternatively, a short kernel can act as implicit device-wide synchronization, responsible only for copying `variables*` to `variables`. This negates the advantage of having a unified kernel since it creates an additional kernel that must be launched separately.

6.2.4 Double-Buffering without Additional Buffer

Double buffering is crucial for implementing a unified kernel; thus, its shortcomings must be addressed. This section describes algorithmic innovation for implementing double buffering without additional buffers. This eliminates the additional memory requirement since the size of each element remains the same, addressing the first shortcoming of the naïve double buffering. It also addresses the second shortcoming of naïve double buffering, reducing the registers and L1 cache pressure.

Each element has buffers for storing `variables` and `contributions`. The `variables` is used for many things, including the volume and flux computation within an element and flux computation for neighboring elements. On the other hand, `contributions` is used solely for integrating an element; no inter-element data transfers involving `contributions`. In addition, once a node inside the element finishes computing the integration and obtains new `variables`, the `contributions` for that node can be discarded. No other nodes will need that `contributions` data.

Therefore, the `contributions` can store the updated variables based on this

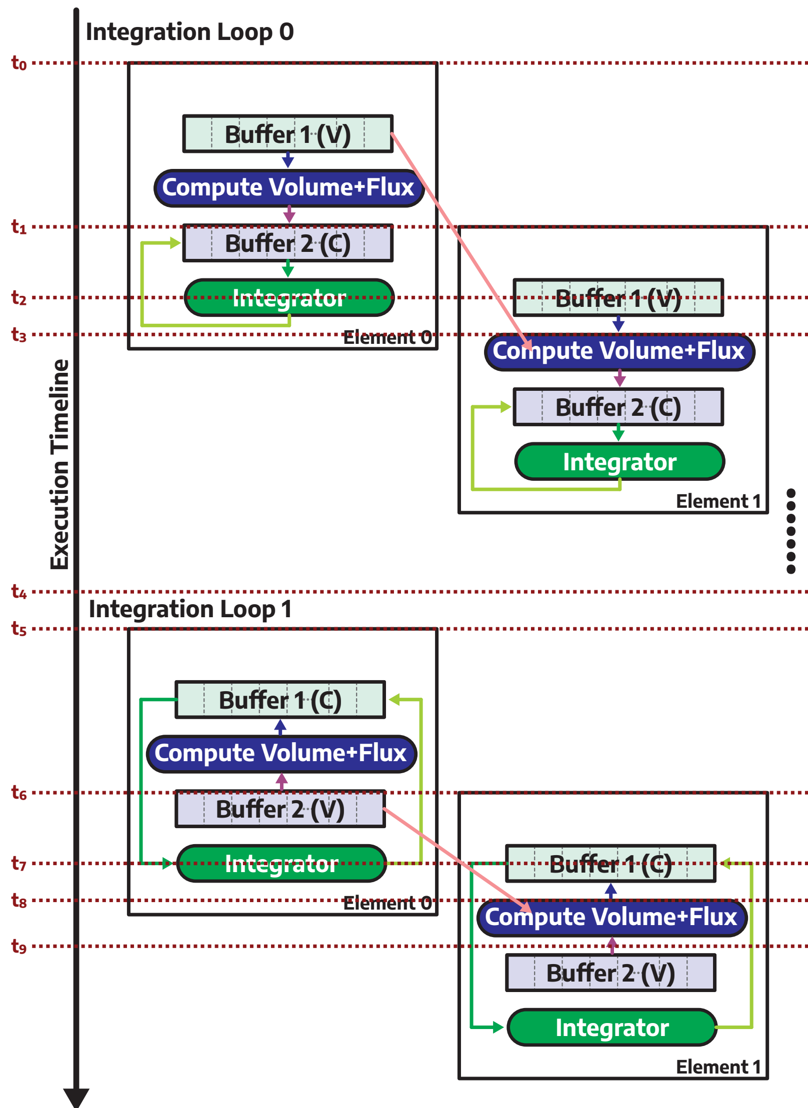


Figure 6.8: The optimized double buffering without needing an additional buffer, thanks to the buffer-swapping strategy. Both `variables` and `contributions` buffers can be swapped interchangeably. The result from computing the integration (i.e., updated variables) is stored in `contributions`, shown at t_2 of Element 0, preserving the contents of `variables` for other elements (i.e., Element 1) to use when computing flux at t_3. Upon finishing the integration loop 0 for all elements, shown at t_4, no additional step or kernel is needed to copy the contents of the buffer. The subsequent integration loop can be performed, as shown at t_5. However, the buffer usage is now reversed, and each element's data flow is slightly modified to accommodate the swapped buffer. The computation on t_5 to t_8 can proceed normally. A flag passed as a kernel argument is used as an agreement on how the buffers are used.

fact. Each node will use its `contributions` to compute the integration and store the result (i.e., updated variables) to the `contributions`. This leaves `variables` untouched, allowing neighboring elements to obtain the correct `variables` data when performing flux computation. After finishing the integration loop or time step, the next integration loop or time step will treat the `contributions` as `variables` and vice-versa. Hence, this technique is called buffer-swapping since both `variables` and `contributions` can be stored interchangeably across these two buffers.

Figure 6.8 illustrates how double buffering with the buffer-swapping technique works. At integration loop 0, all elements agree to use Buffer 1 to store `variables` and Buffer 2 to store `contributions`. The computation proceeds as usual for Element 0, starting at t_0, except when computing the integration at t_2, the updated variables are directly stored inside Buffer 2, as indicated by the green arrow. Likewise, Element 1 performs its computation normally, starting at t_1. It obtains neighboring element data from Buffer 1 on Element 0 containing `variables` for computing flux at t_3.

After all elements finish the Integration Loop 0, the subsequent integration loop can be started as soon as possible, at t_5, without copying the contents between buffers. The t_6 to t_8 follows the same execution as t_1 to t_3, except, now, all elements agree to use Buffer 1 to store `contributions` and Buffer 2 to store `variables`. Therefore, the data flow in each element is changed, as shown by the arrow in Figure 6.8. It is slightly modified from the original data flow shown in Figure 4.10. This modification eliminates the third shortcoming of naïve double buffering, which requires a separate kernel or device-wide synchronization barrier to correctly copy data between the two buffers. Since each integration loop and time step involves launching the simulation kernel, shown in Figures 4.9, 5.4, and 6.5, the flag indicating the agreement for all elements regarding with the buffer usage can be passed as parameter to the GPU kernel. Based on this flag, the correct buffer can be chosen to perform operations and update the ghost buffer for multi-GPU runs.

6.2.5 Unified Kernel Implementation

It is time to implement the unified kernel after solving the crucial issue of implementing a unified kernel using the optimized double buffering to avoid intra- and inter-element data hazards. This includes implementing the buffer-swapping mechanism based on the flag, adjusting the parallelism extraction for the integration kernel, utilizing shared memory, and implementing ghost exchange for multi-GPU support.

6.2.5.1 Implementing Buffer-Swapping Mechanism

All elements must agree on using the two buffers: which buffer stores `variables` and which buffer stores `contributions`. A flag is used to indicate the buffer-swapping mechanism. The flag consists of a boolean value; a value of 0 means there is no change in the position of the buffer (i.e., following the original data flow of the element, shown in Figure 4.10). On the other hand, a value of 1 indicates both buffers are swapped, and thus, the data flow of the element is slightly modified. The flag is inverted every time the integration loop finishes and is given to the simulation kernel as an argument.

Based on this flag, the `variables` and `contributions` can be mapped into the correct buffers. Two uninitialized pointers represent the `contributions` and the `variables`. Then, depending on the state of the flag, the pointers are assigned with the address of the correct buffers. For example, if Buffer 1 stores the `variables`, then the pointer to `variables` is assigned with the address of Buffer 1. It is the same thing for the pointer to `contributions`, assigned with the address of Buffer 2. If the buffer is swapped, then the assignments are swapped. The subsequent operations only need to use the pointer to `variables` and the pointer to `contributions` to access the correct data, eliminating the need for changing the location of every operand and resultants programmatically.

6.2.5.2 Adjusting Parallelism of Integration

As described in Section 5.3.4, the integration kernel extracts parallelism at the element-level, node-level, and variable level, with the total number of threads equal to $N_{element} \times N_{nodes_per_element} \times N_{variables}$. However, the fused volume and flux kernel described in Section 5.4.1 extracts parallelism at the element level and node-level, with a total number of threads equal to $N_{element} \times N_{nodes_per_element}$.

Adopting the strategy used in kernel fusion as described in Section 5.4.1, one thread is dedicated to handling one node throughout the kernel execution. Therefore, the parallelism for the integration must be reduced to element-level and node-level, resulting in a total thread of $N_{element} \times N_{nodes_per_element}$. If Node-Tiling is used (Section 6.1.3), then the total thread is $N_{element} \times N_{nodes_per_tile}$.

6.2.5.3 Shared Memory Utilization with Buffer-Swapping

Using shared memory for storing `variables` and `contributions` is highly recommended, especially when the unified kernel uses node-tiling execution. This eliminates the pointers assignments, as described in Section 6.2.5.1 and allows for holding the important element's data inside the on-chip memory of an SM, eliminating the risk of being evicted to L2 cache or off-chip HBM2 memory.

Using shared memory for storing the `variables` is done by copying the correct buffer, based on the flag, to the array stored in shared memory. All subsequent reads to `variables` can be served quickly through the shared memory. On the other hand, using the shared memory for storing `contributions` is done by directly storing and accumulating the contributions in shared memory. The integration will store the updated variables into the `contributions` stored in the shared memory. At the end of the kernel, the contents are copied to the correct buffer used to store `contributions` in global memory.

6.2.5.4 Ghost Exchanges for Multi-GPU Support

The involvement of the ghost exchange slightly complicates the unified kernel implementation. Depending on whether the neighboring elements are ghost elements, the flux computation may not be finished for all nodes inside an element. Nodes that rely on ghost elements to compute the flux contributions must wait for ghost exchange to finish, preventing them from directly executing the integration to update the `variables`. The flux computation is then performed as external flux followed by integration, as shown in Figure 6.5.

Figure 6.9 shows an example of the execution flow for each node within an element when running the unified kernel of volume, internal flux, and integration. The volume and flux parts are exactly similar to the fused and internal flux described in Section 5.4.1. At the beginning of execution, all nodes are assigned with flag F initialized to 1, assuming all flux computation has been completed. All nodes perform the volume contribution computation. However, not all nodes perform flux computation on every face. Each node can quickly determine its position on the element's faces by utilizing the node lookup described in Section 5.3.3.3. For flux computation on Face 1, the neighboring element is a ghost, and thus, the flux cannot be computed as it needs to wait until ghost exchange is performed. In this case, the flux computation is skipped, and the flag F for the nodes on that face is changed to 0. After visiting all of the faces, the integration is performed. Only nodes with flag F equal to 1 can perform the integration. Otherwise, it is skipped.

After the ghost exchange, nodes that depend on ghost elements to perform the flux computation are revisited by launching the fused external flux and integration kernel. It has a similar execution flow as the unified kernel illustrated in Figure 6.9, except no volume computation and all nodes are assigned with the flag F equal to 0, assuming integration has been done, and thus no need to perform the integration. Upon visiting every face of the element, only nodes that are located on the face and have a ghost element as their neighbor perform their flux computation. For these

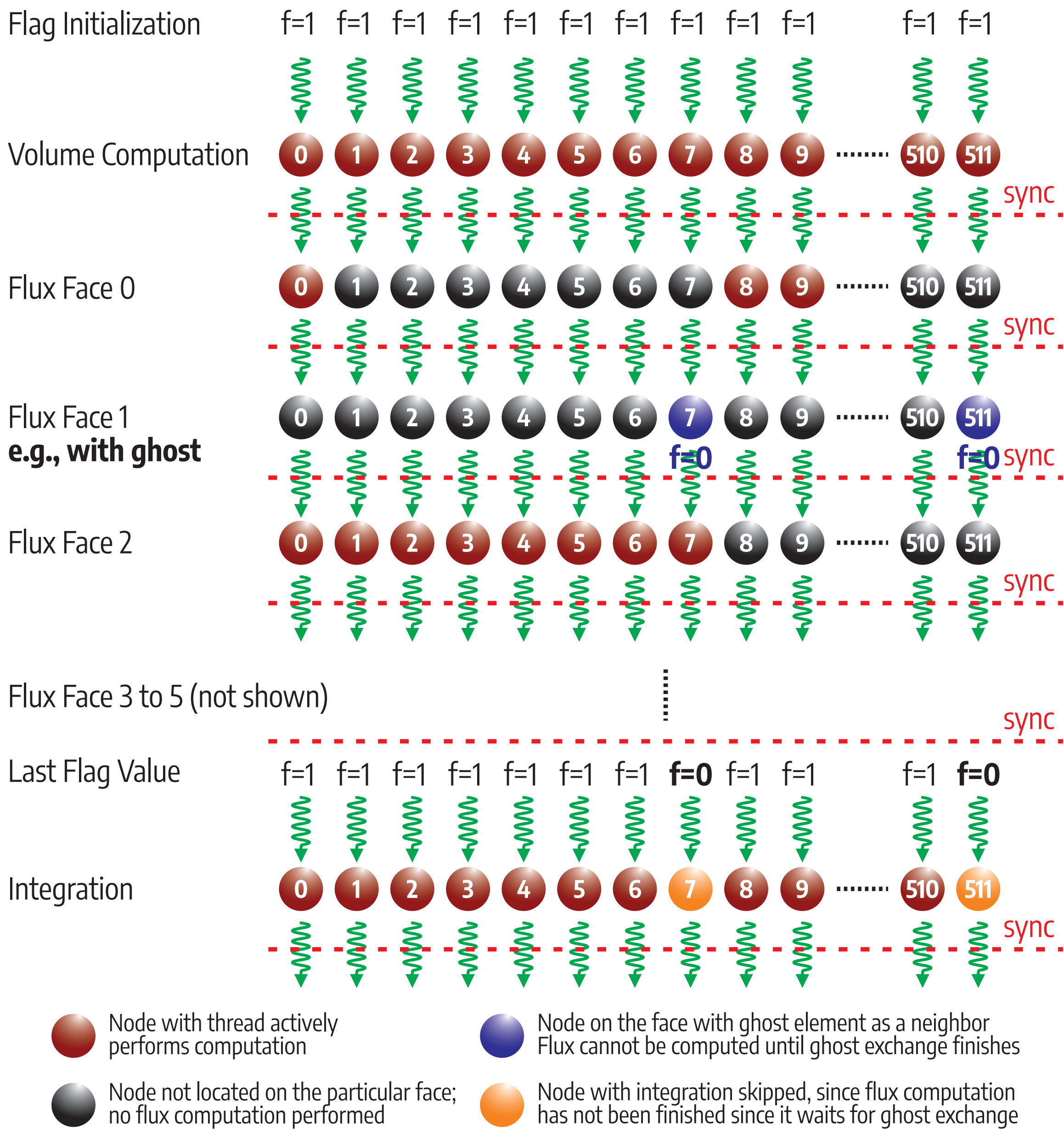


Figure 6.9: The execution flow of the unified kernel, showing the operations done by each node in an element. Assuming a 512-node element where the neighboring element at Face 1 is a ghost. Initially, all nodes are given a flag F initialized as 1, meaning flux computation is finished for a particular node. During flux computation on Face 0, not all nodes are located on that face; thus, some skip the flux computation and are shown as black nodes. For Face 1, flux contribution for nodes 7 and 511 cannot be computed since it needs the ghost element, and thus, flag F becomes 0 for these two nodes. Only nodes with flag F equal to 1 can run the integration.

nodes, the flag F is changed to 1, forcing the integration to be performed.

6.3 Face-Nodes-Only Ghost Exchange

In large-scale wave simulations, inter-node communication remains the key bottleneck since the inter-node interconnect is often the weakest link in computing clusters. The effort of reducing the inter-node communication overhead has been made previously in Section 5.6.2 by reducing the amount of data that needs to be exchanged between GPUs. It is done by sending only the required data for the neighboring elements to perform flux computation: the `variables` and `materials`. This results in reducing the ghost elements' size to 27% of the standard element, translating to 73% communication overhead reduction. In addition, this ghost exchange optimization does not incur additional compute overhead or affect the wave simulations' numerical accuracy. It is worth noting that having two different data types, for elements and ghost elements, is impossible with `p4est` since both must have the same data types.

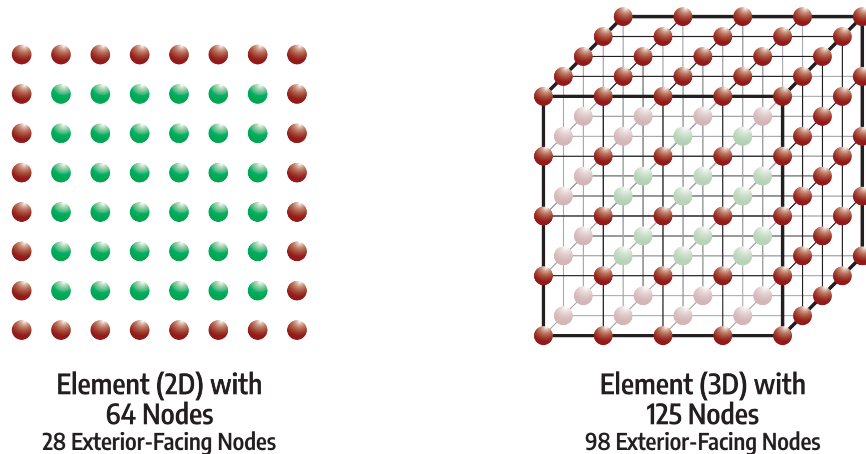


Figure 6.10: Example of the nodes located on the faces of elements (i.e., exterior-facing nodes), as drawn in red dots. The interior nodes, drawn in green, do not need to be exchanged between GPUs during ghost exchange. Only the exterior-facing nodes are required for ghost exchange.

This section discusses further optimization of ghost exchange by only sending data of the nodes located on the faces of the elements. In other words, the nodes

located in the interior of the elements are not to be exchanged. Figure 6.10 illustrates the nodes of the elements where the red and green represent exterior-facing and interior nodes, respectively. With 512 nodes per element, only 296 nodes need to be sent during the ghost exchange. However, unlike ghost exchange optimization described in Section 5.6.2, this optimization incurs additional compute overhead without impacting numerical accuracy.

6.3.1 Modifying `ElementDataBase` Structure

The first step is to modify the `ElementDataBase` to reduce its size by removing unnecessary members and shrinking the size of `variables` array to only contain the exterior-facing nodes. For an element with 512 nodes, only 296 nodes are stored inside the new data structure, which is called `ElementDataBaseGhostV2`. For acoustic wave simulations with four variables (Figure 2.1), the `variables` only contains 1,184 items instead of 2,048 items. For elastic wave simulations with nine variables (Figure 2.1), the `variables` only contains 2,664 items instead of 4,608 items.

Problem	**Precision**	**# Nodes per Element**	**Size (Bytes)**		
			Standard	**GhostV1**	**GhostV2**
Acoustic	FP32	512	32,832	8,200	4,744
Acoustic	FP64	512	65,640	16,400	9,488
Elastic	FP32	512	63,556	18,444	10,668
Elastic	FP64	512	127,088	36,888	21,336

Table 6.1: The Standard, GhostV1, and GhostV2 Element Size for 512-node Elements. The GhostV1 is discussed in Section 5.6.2. GhostV2 will no longer have 512 nodes inside each element since it only contains face nodes.

The `materials` remains the same since it only includes two and three data for acoustic and elastic wave simulations. However, if each node has its material properties (i.e., `NNODE_MATERIAL_1D`=`NNODE_1D`, as defined in Appendix B.5.3) then it can also be shrunk to only contain the materials from exterior-facing nodes. Table 6.1 shows the size of the `ElementDataBaseGhostV2` as a result of this modification, compared to the standard `ElementDataBase` and previous `ElementDataBaseGhostV1`.

This new data structure, `ElementDataBaseGhostV2` for ghost exchange is 42% smaller than the one implemented in Section 5.6.2, `ElementDataBaseGhostV1`, and is 84% smaller than the standard `ElementDataBase`. A new compile-time configuration, `GPU_GHOST_ELEMENT`, is introduced to allow choosing the type of data structure used for ghost elements, which is discussed in Appendix B.5.5.

6.3.2 Implementation Challenge

Shrinking the number of items inside the `variables` results in difficulty accessing the variables for a particular node inside the array. The `variables` no longer follows the node indexing scheme defined in Section 4.1.3. Figure 6.11 illustrates this challenge, where the red items indicate the exterior-facing nodes while the green items indicate the interior nodes.

The `variables` array in `ElementDataBaseGhostV1` still has the same length as the `variables` array in `ElementDataBase`, and thus, the indexing can be done straightforward. However, since it also contains the interior nodes (green), only the exterior-facing nodes (red) are accessed. On the other hand, the `variables` array in `ElementDataBaseGhostV2` no longer contains interior nodes, which breaks the index mapping scheme from `ElementDataBase`. The interior nodes in `ElementDataBase` will have undefined mapping. In contrast, the exterior-facing nodes will require some offset to map into the correct index on `variables` array of `ElementDataBaseGhostV2`.

6.3.3 Array Index Mapping Strategy

This section discusses the approach to overcome the node mapping challenge, described in Section 6.3.2 as a result of shrinking the `variables` only to contain the exterior-facing nodes. There are two solutions for the problem: computing the node offset through an algorithm and using a look-up table to find the correct mapping of the node. The former incurs some compute overhead, while the latter incurs additional memory space and access overhead.

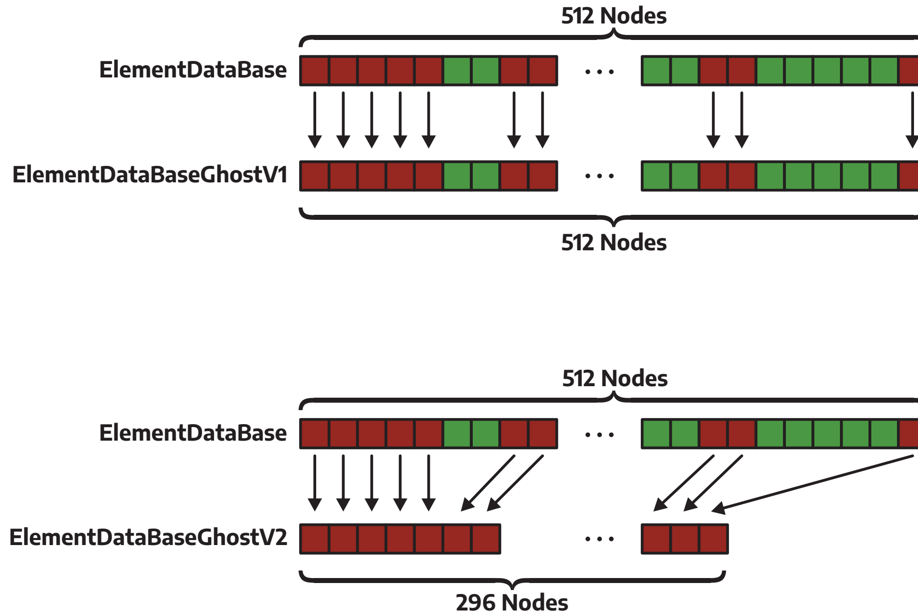


Figure 6.11: The indexing challenge for accessing the node data inside `variables` in `ElementDataBaseGhostV2`. The red items indicate exterior-facing nodes, while the green items indicate interior nodes. The previous implementation of ghost structure, `ElementDataBaseGhostV1` uses the same length of `variables` array as the standard `ElementDataBase`, making the indexing done straightforward. On the other hand, `variables` in `ElementDataBaseGhostV2` only contains the exterior-facing nodes, making indexing more difficult.

6.3.3.1 Calculating Offset Index for Node Mapping

The first strategy to solve the node mapping problem is calculating the correct offset from the node index used in `ElementDataBase` to obtain the correct node index for `ElementDataBaseGhost`. Algorithm 6 shows the algorithm for calculating the node offset in an element with 512 nodes. Note that different element configurations (i.e., elements with different numbers of nodes) will most likely require different algorithms to compute the node offset.

The algorithm only requires integer computation with no branches or loop operations. Starting from the Turing architecture, the SM can execute integer and floating-point operations concurrently, as discussed by NVIDIA Corporation (2018, 2020a). Since almost all operations in wave simulations are floating-point arithmetic,

Algorithm 6: Calculate the offset for indexing into the face node on `ElementDataBaseGhostV2` given the face node index of `ElementDataBase`

Defined:
Number of Nodes in Each Direction (`NNODE_1D`) $\mathcal{N}$

Inputs:
Face Node Index from `ElementDataBase` $\mathbf{P}$

Outputs:
Face Offset Index to `ElementDataBaseGhostV2` $\mathbf{O}$

Main:

1. $i \leftarrow \mathbf{P} \bmod \mathcal{N}$
2. $j \leftarrow (\mathbf{P} \text{ div } \mathcal{N}) \bmod \mathcal{N}$
3. $k \leftarrow (\mathbf{P} \text{ div } (\mathcal{N} \times \mathcal{N})) \bmod \mathcal{N}$
4. $k_0 \leftarrow 1 - (k \text{ and } 1)$
5. $j_0 \leftarrow 1 - (j \text{ and } 1)$
6. $i_t \leftarrow i \text{ div } (\mathcal{N} - 1)$
7. $j_m \leftarrow (j \bmod (\mathcal{N} - 1)) \text{ and } 1$
8. $k_m \leftarrow (k \bmod (\mathcal{N} - 1)) \text{ and } 1$
9. $\mathbf{O}_k \leftarrow (k - 1 + k_0) \times (\mathcal{N} - 2) \times (\mathcal{N} - 2)$
10. $\mathbf{O}_j \leftarrow (j - 1 + j_0) \times (\mathcal{N} - 2)$
11. $\mathbf{O}_i \leftarrow i_t \times (\mathcal{N} - 2)$
12. $\mathbf{O} \leftarrow \mathbf{O}_k + (\mathbf{O}_j \times k_m) + (\mathbf{O}_i \times k_m \times j_m)$

this algorithm can run concurrently on the integer ALUs without adding more work to the floating-point ALUs.

The algorithm needs the `NNODE_1D` (Section 4.1.2) defined correctly as $\mathcal{N}$. Then, it asks for the node index from `ElementDataBase` as an input $\mathbf{P}$. Note that this algorithm only accepts the index from nodes located at the faces of the element; giving the index of interior nodes will result in an incorrect offset since there is no defined mapping of `variables` between `ElementDataBase` to `ElementDataBaseGhostV2` for interior nodes, as shown in Figure 6.11. Next, the algorithm will compute the node's position in 3D space (i.e., i, j, k) and additional information obtained from the coordinate (i.e., k_0, j_0, i_t, j_m, k_m). Finally, the algorithm calculates the offset of the node

position in each axis (i.e., $\mathbf{O}_x, \mathbf{O}_y, \mathbf{O}_z$) to finally calculate the node index offset for indexing into the shrunk `variables` array.

Original Index (ElementDataBase)		Offset		Final Index (ElementDataBaseGhostV2)
511	-	216	=	295
263	-	108	=	155
133	-	36	=	97
0	-	0	=	0
78	-	0		Not applicable (interior node)

Figure 6.12: An example of how to use the index offset generated by Algorithm 6. If an external-facing node index is given, it outputs the offset to subtract the original index, resulting in the correct index to the `variables` inside `ElementDataBaseGhostV2`. However, if an interior node index is given, the resulting offset is invalid and should be discarded.

Figure 6.12 shows an example of using this algorithm to calculate the offset. The algorithm will output an offset for a given face node index on `ElementDataBase`. This offset is then used to subtract the given face node index, resulting in the correct index for accessing `variables` in `ElementDataBaseGhostV2`. However, if the interior node index is given, the algorithm result is invalid and should be discarded.

6.3.3.2 Using Look-up Table for Node Mapping

The second approach uses a look-up table (LUT) to precompute and store the node index mapping. LUTs have been used on a couple of occasions in this dissertation: neighbor look-up (Section 5.1.2.2) and node look-up (Section 5.3.3.3).

With special memory inside a GPU called constant memory (Section 2.6.3.1), the LUT can be accessed very quickly. For the case of implementing LUT for node mapping, a 16-bit integer must be used since an 8-bit integer is not sufficient to store index 0 to 512. This results in LUT with the size of 1 KB, which is sufficiently small. However, every element configuration (i.e., the element with a different number of nodes) will have its own LUT.

6.3.4 Modifying Kernels to Support New Ghost Elements

After taking care of the node mapping to allow accessing the new shrunk `variables` correctly, the next step is to call the mapping function every time access to the `variables` of a ghost element is needed. Previous implementation, the `ElementDataBaseGhostV1`, does not require modification from the kernel side as the kernel can access the `variables` as usual.

Two kernels are affected by the change: the flux kernel and the kernel used to update the ghost buffer (Section 5.6.1). Since only the external-facing nodes are involved in flux computation, the modification to the kernel is done by simply subtracting the original node index with the offset, resulting in the correct index to `ElementDataBaseGhostV2`. On the other hand, for updating the ghost buffer, it is no longer possible to copy the whole `variables`; instead, it needs to check whether it is an exterior-facing node before copying the data one by one in a loop. This is also why such ghost exchange optimization cannot be implemented in `p4est` since it handles the ghost buffer internally, requiring modification from the library.

6.4 Reduced-Precision Ghost Exchange

Lower precision floating-point format has become increasingly common for use in machine-learning applications. The smaller data types reduce the pressure on memory capacity and bandwidth requirements, giving more arithmetic intensity (i.e., more FLOPs/byte) and improving performance. Modern computing hardware,

such as graphics processing units (GPUs), integrate specialized functional units that provide high computation throughput for smaller data types, as discussed in Section 2.6.4. However, using lower-precision data types may cause numerical accuracy and stability issues for applications where high-precision arithmetic is crucial, such as in high-performance scientific applications, including seismic wave simulations. In this case, it may not be desirable to perform all calculations using lower-precision arithmetic. Instead, mixed-precision arithmetic is commonly used, as discussed in Section 2.6.5.

This section discusses adopting the mixed-precision strategy for reducing inter-node communication overhead by using reduced precision for ghost exchange while keeping all local computations at double precision[1]. By using lower precision data types, such as single- and half-precision, the data volume can be lowered by 50% and 25%, respectively, compared to when the ghost exchange is performed using double-precision data types. However, this optimization may cost some numerical accuracy, unlike ghost exchange optimization described in Section 5.6.2. Thus, the effect of reduced-precision ghost exchange on numerical stability should be evaluated carefully.

6.4.1 Multi-Precision Strategy

Multi-precision or mixed-precision computation is performed in machine learning applications as discussed in Section 2.6.5. Most parts of the machine learning model will be represented using the lower precision data types (e.g., half-precision) to reduce the memory consumption and bandwidth requirements. Higher precision data types (e.g., single-precision) will be used for critical network parts where ac-

[1]My collaborator from ExxonMobil Technology and Engineering Company, Arash Fathi, proposed the idea of reduced-precision ghost exchange to reduce the amount of data volume during ghost exchange after we learned together that the inter-node communication limits the overall simulation performance. Based on his idea, I implemented and evaluated the reduced-precision ghost exchange on GPUs. More sophisticated experiments are being performed by collaborating with him at the time of writing, eventually leading to future publication.

curacy is essential, such as storing the optimizer states and accumulating gradients. This strategy works well for training machine learning models as it yields satisfactory model accuracy, lowers memory usage, and reduces the time needed to train the models thanks to specialized functional units capable of accelerating low-precision arithmetic.

On the other hand, in high-performance scientific applications, arithmetic precision is the most important, as it can affect numerical accuracy and stability. While machine learning applications are sufficient to run at single-precision or lower, scientific applications demand double-precision arithmetic. Therefore, instead of keeping the majority of arithmetic operations at lower precision, the majority must be kept at higher precision; this is the opposite of how mixed precision arithmetic is done in machine learning applications. In the case of the wave simulation application discussed in this dissertation, only the ghost exchange and the computation of external flux contribution (i.e., flux for faces interfacing a ghost element) is done in lower precision (e.g., single-precision or half-precision). The other operations are performed at higher precision (e.g., double-precision or single-precision). Although there are no exact rules on how much of the arithmetic operations can be performed with lower precision without losing too much numerical accuracy, causing instability, the lower proportion is the better.

Figure 6.13 shows the strategy of using mixed precision arithmetic for ghost exchanges. It outlines three data types for three purposes: storing the data in memory, performing external flux computation, and exchanging it through ghost exchange. The `DDD` and `SSS` means double precision and single precision are used for all parts of the wave simulations, respectively. This is the standard wave simulation application configuration, which only uses one precision, as explained in Chapter 5.

With multi-precision implementation, other variations can be used. The `DDS` and `DDH` keep all computation in double-precision arithmetic while only the data for ghost exchange is stored in single-precision and half-precision formats, respectively.

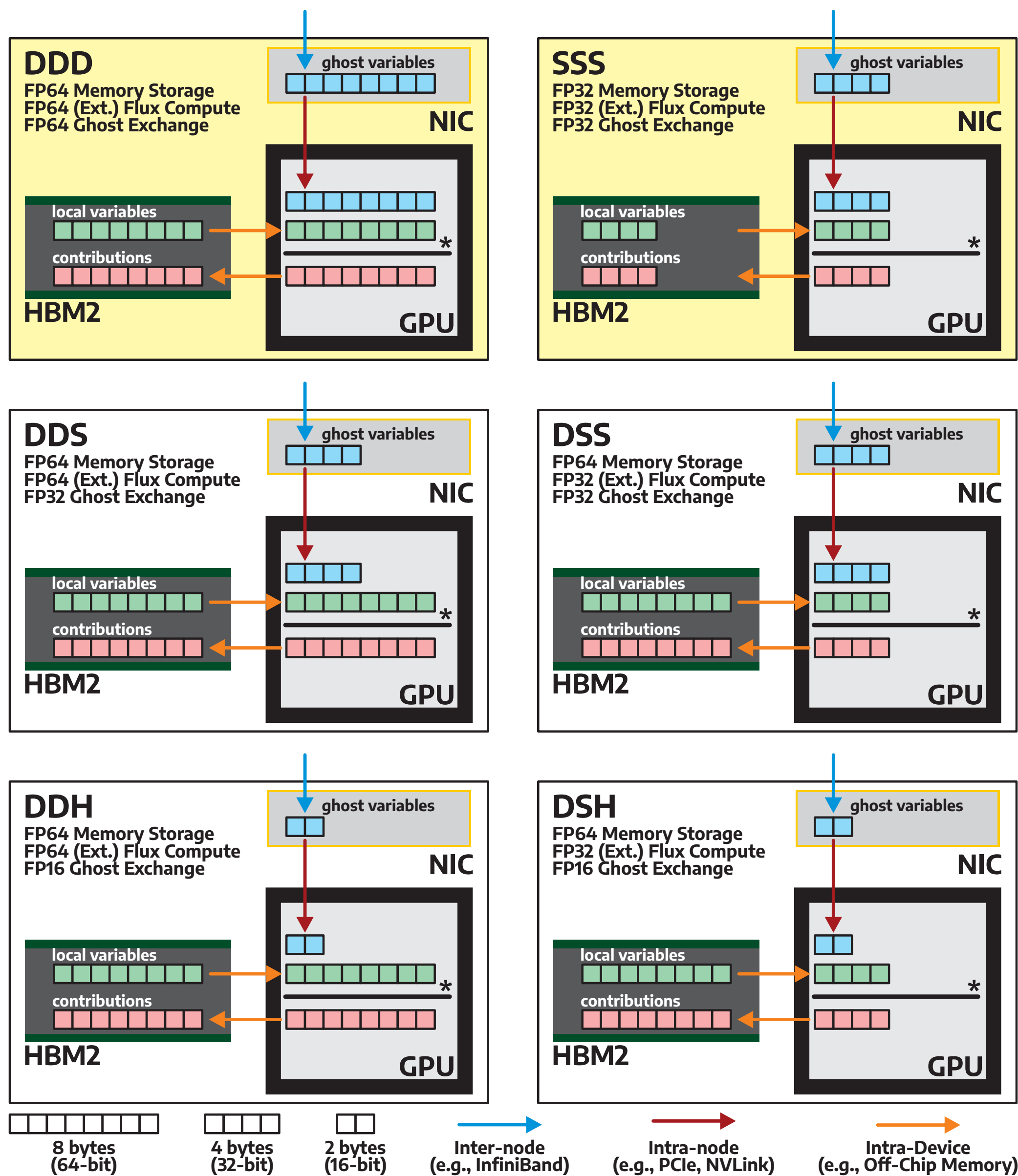


Figure 6.13: Several mixed-precision strategies for implementing reduced-precision ghost exchanges. The `DDD` and `SSS` are the standard ghost exchanges used in Section 5.6, where both the main computation and the data transfers use the same precision: double-precision and single-precision, respectively. These are not an exhaustive list of strategies. Note that in Longhorn (Section 3.2.2), the NIC must traverse through PCIe, CPU, and NVLink to reach the GPU, while in Lonestar6 (Section 3.2.3), the NIC will need to go through PCIe, CPU, and PCIe to reach the GPU.

On the other hand, DSS and DSH use single-precision arithmetic to compute the external flux only, keeping the rest of the computation in double-precision. At the same time, the ghost exchange is performed in single-precision and half-precision, respectively. This is not an exhaustive list since other combinations are possible, such as DHH, SSH, and SHH. In addition, floating-point formats outside the IEEE 754 standard exist, such as BFloat16 described by Wang and Kanwar (2019), which can be other options. These are not explored in this dissertation.

6.4.2 Code Implementation

Implementation of reduced-precision ghost exchange does not require significant code changes. Two compile-time configurations are added to make it easier to configure different data types without explicitly changing the floating-point precision on the source code one by one. The GPU_GHOST_CPRECISION defines the floating-point data type for computing the external flux involving the ghost elements. In contrast, the GPU_GHOST_EPRECISION represents the floating-point data type used for data transmitted during ghost exchange. These two options are described in detail in Appendix B.5.5. The original PRECISION is still used to control the precision for all other computations.

6.4.2.1 Ghost Element Data Structure Modification

The next modification is to define the ghost element data structure. Instead of following the main precision defined through the compile-time configuration PRECISION, it follows the compile-time configuration GPU_GHOST_EPRECISION. This cannot be accomplished using p4est since it always uses the same data structure using the same data type for both standard elements and ghost elements unless users modify the source codes under the hood. Table 6.2 shows the size of the ghost element data when instantiating with reduced precision different from what the main precision is used. Note that using higher precision for the ghost element than the

main precision makes no sense; thus, this combination is omitted.

Problem	Main Precision	Ghost Precision	# Nodes per Element	Size (Bytes)		
				Standard	GhostV1	GhostV2
Acoustic	FP32	FP16	512	32,832	4,100	2,372
Acoustic	FP32	FP32	512	32,832	8,200	4,744
Acoustic	FP64	FP16	512	65,640	4,100	2,372
Acoustic	FP64	FP32	512	65,640	8,200	4,744
Acoustic	FP64	FP64	512	65,640	16,400	9,488
Elastic	FP32	FP16	512	63,556	9,224	5,336
Elastic	FP32	FP32	512	63,556	18,444	10,668
Elastic	FP64	FP16	512	127,088	9,224	5,336
Elastic	FP64	FP32	512	127,088	18,444	10,668
Elastic	FP64	FP64	512	127,088	36,888	21,336

Table 6.2: The Standard, GhostV1, and GhostV2 Element Size for 512-node Elements with Reduced-Precision Ghost Exchanges. The main precision is defined through compile-time configuration `PRECISION` while the ghost precision is defined through compile-time configuration `GPU_GHOST_EPRECISION`.

6.4.2.2 Removing Code-Clutter Due to Type-Casting

Another problem is writing the codes supporting multi-precision arithmetic while maintaining cleanliness and making debugging easy. Often, the data must be converted to another precision during the arithmetic operations, such as when the ghost computes precision (`GPU_GHOST_CPRECISION`) different than the main precision `PRECISION`. This is usually done through explicit type-casting, where the target data type is placed before the operands or operations. However, too much usage of explicit type-casting in the codes creates clutter, making the codes less readable and more difficult to debug, as shown in Figure 6.14.

The solution to writing multi-precision codes while not introducing clutter to the code is by defining a C++ class along with overloaded operators. A C++ class called `ghost_mp` is declared along with its constructors and overloaded operators. The overloaded operators include the type-casting to `real_cghost_t` for the second operand of a different type to be computed with the data from the ghost element. Operators that are overloaded for various precision for the second operand include

One precision code

```
real_t vn_0_bracket = (vt_0[selfNodeID + axis * NNODE]
                      - vt_1[neighborNodeID + axis * NNODE]) * selfFaceNormal;
```

Multi-precision code

```
real_cghost_t vn_0_bracket = (real_cghost_t)
                             (((real_cghost_t)vt_0[selfNodeID + axis * NNODE]
                             -(real_cghost_t) vt_1[neighborNodeID + axis * NNODE_GHOST])
                             * (real_cghost_t)selfFaceNormal);
```

Figure 6.14: The C++ codes for one precision support (top) and multi-precision support (bottom). The multi-precision codes use type-casting to explicitly convert one precision to another, making the code cluttered, less readable, and harder to debug. The `real_t` is a data type defined by compile-time configuration `PRECISION`, while `real_cghost_t` is a data type defined by compile-time configuration `GPU_GHOST_CPRECISION`.

addition (+), subtraction (−), multiplication (×), division (/), and assignment (=). A square root operator ($\sqrt{}$) is also defined across different precision. Figure 6.15 shows some parts of the multi-precision ghost class declaration and the usage example. The decorator `__device__` indicates that the class members are accessible from the GPU devices. Using this class, the arithmetic operations can be expressed as the regular one-precision arithmetic, significantly improving code readability.

6.5 Early Exploration of Partial Ghost Exchange

Previous solutions for reducing the communication overhead associated with the ghost exchange are reducing the data volume. All solutions send all `variables` through the ghost exchange: four for acoustic and nine for elastic wave simulations. This section explores the idea of partial ghost exchange[2] by sending only parts of

[2]My collaborator from ExxonMobil Technology and Engineering Company, Arash Fathi, proposed the idea of partial ghost exchange to reduce the data volume during ghost exchange after we learned together that the inter-node communication limits the overall simulation performance. I collaborated with him to evaluate this idea, which is still in its infancy and serves only as an early exploration in this dissertation. More sophisticated experiments are being performed by collaborating with him, eventually leading to future publication.

Multi-precision Ghost Class and Overloaded Operators

```
class ghost_mp
{
  public:
  // constructor without argument
  __device__ ghost_mp() { val = (real_cghost_t) 0.0_p; }
  // constructor with argument of double
  __device__ ghost_mp(double input) { val = (real_cghost_t) input; }
  ...
  // operator + overloading
  __device__ ghost_mp operator+(const ghost_mp &input){return val+input.val;           }
  __device__ ghost_mp operator+(const double input)   {return val+(real_cghost_t) input;}
  ...
  // operator - overloading
  __device__ ghost_mp operator-(const ghost_mp &input){return val-input.val;           }
  __device__ ghost_mp operator-(const double input)   {return val-(real_cghost_t) input;}
  ...
  // operator * overloading
  __device__ ghost_mp operator*(const ghost_mp &input){return val*input.val;           }
  __device__ ghost_mp operator*(const double input)   {return val*(real_cghost_t) input;}
  ...
  // operator / overloading
  __device__ ghost_mp operator/(const ghost_mp &input){return val/input.val;           }
  __device__ ghost_mp operator/(const double input)   {return val/(real_cghost_t) input;}
  ...
  // operator assignment
  __device__ void operator=(const ghost_mp &input)    {val = input.val;           }
  __device__ void operator=(const double input)       {val = (real_cghost_t) input;}
  ...
}
```

Usage Example

```
ghost_mp vt_self     = vt_0[selfNodeID + axis * NNODE];
ghost_mp vt_neighbor = vt_1[neighborNodeID + axis * NNODE_GHOST];
ghost_mp vn_0_bracket= ((vt_self - vt_neighbor) * selfFaceNormal);
```

Figure 6.15: Parts of C++ class declaration for multi-precision ghost and the overloaded operators for various precisions of the second operand. Using this class, the arithmetic involving different types of precisions for ghost and local data can be expressed in cleaner, readable codes. The decorator `__device__` indicates the class members are implemented for GPU devices.

`variables` and approximating the rest on the receiving side. For example, in elastic wave simulations, only the three particle velocity variables, $\mathbf{v}_x$, $\mathbf{v}_y$, and $\mathbf{v}_z$, are transferred during the ghost exchange. On the other hand, the other six variables related to stress values ($\mathcal{S}$) are approximated by the receiver using local computations.

$$(\mathbf{v}, \mathcal{S}) \rightarrow (\mathbf{v}) \tag{6.1}$$

Equation (6.1) illustrates the partial ghost exchange, where $(\mathbf{v}, \mathcal{S})$ indicates velocity and stress values residing on the sender node, $(\mathbf{v})$ indicates velocity values

received at the receiving node, and $\rightarrow$ denotes the ghost exchange. In this case, only $\frac{1}{3}$ data are sent, reducing the volume of data by 66.68%. However, while it can reduce the ghost exchange overhead, this optimization generates additional computation overhead to approximate the stress $\mathcal{S}$ values.

Two strategies are explored in this section to approximate the stress ($\mathcal{S}$) values. The approximation strategy must maintain numerical stability with minimal impact on solution accuracy. Therefore, before implementing them on the GPUs, they are investigated by developing the CPU code to run the experiments and observing their stability and accuracy.

6.5.1 Approximation of Stress Values by Computing Displacements

Using the combined constitutive and kinematics equation, as shown in Equation (6.2), stress values ($\mathcal{S}$) can be computed. The u denotes the displacement vector and can be computed by the receiving node by integrating the velocity vector as shown in Equation (6.3).

$$\mathcal{S} = \mu \left[\nabla u + (\nabla u)^T \right] + \lambda (\text{div } u)\ \mathcal{I} \tag{6.2}$$

$$u(x,t) = u(x,t_0) + \int_{t_0}^{t} \mathbf{v}(x,\tau) d\tau \tag{6.3}$$

However, upon developing the CPU codes and running the experiments, it was observed that this strategy yields unstable formulation. Therefore, further development of this strategy as a partial ghost exchange method for GPU acceleration is not pursued.

6.5.2 Approximation of Stress Values by Integrating Constitutive and Kinematics Equation

Another strategy worth exploring is to approximate the stress values ($\mathcal{S}$) by integrating the time-differentiated constitutive and kinematics equation directly at the

receiving node, as shown in Equation (6.4). The $\mathcal{S}(x,t)$ indicates the approximated stress values, and $\mathcal{S}(x,t_0)$ denotes the corresponding initial values.

$$\mathcal{S}(x,t) = \mathcal{S}(x,t_0) + \int_{t_0}^{t} \left(\mu \left[\nabla \mathbf{v} + (\nabla \mathbf{v})^T\right] + \lambda(\text{div}\mathbf{v})\ \mathcal{I}\right) d\tau \tag{6.4}$$

An advantage of using Equation (6.4) over the approach discussed in Section 6.5.1 is the ability to reset the stress values after a couple of iterations if required. All variables $(\mathbf{v}, \mathcal{S})$ are transmitted through the ghost exchange during the reset, as shown in Equation (6.5), like the complete ghost exchanges. Then, the received stress values become the new initial values $\mathcal{S}(x,t_0)$. Therefore, the effective volume of data transmitted during the ghost exchange depends on how often the resets must be performed to maintain numerical stability.

$$(\mathbf{v}, \mathcal{S}) \rightarrow (\mathbf{v}, \mathcal{S}) \tag{6.5}$$

6.6 Intra-Device Communication Performance Evaluation

This section evaluates the intra-device communication-reducing strategies discussed in Sections 6.1 and 6.2: the node-tiling execution and the unified kernel, respectively. By applying these strategies along with the optimizations discussed in Section 5.4, there are several GPU code flavors (i.e., kernel set) that can be developed, as discussed in Section 6.6.1. Following the performance evaluation discussed in Section 6.6.2, the simulation runtime for each GPU code flavor on single NVIDIA Tesla V100 GPU is compared in Section 6.6.2.

6.6.1 GPU Code Flavors

The optimization strategies discussed in Section 5.4 combined with the intra-device communication-reducing strategies discussed in Sections 6.1 and 6.2 results in several combinations of GPU code flavors, as summarized in Table 6.3. The `GPU_base` is the basic GPU codes described in Section 5.3 while the `GPU_fl` and `GPU_fls` are

taken from the optimized GPU kernels discussed in Sections 5.4.1 to 5.4.3. Note that in GPU_fls, only contributions are stored inside the shared memory of the GPU.

Kernel Set Name	Optimization and Intra-Device Communication-Reducing Strategy	Described in Section
GPU_base	Basic (base)	5.3
GPU_fl	Fused Volume and Flux with LUT-based Node Look-up (fl)	5.4.1
GPU_fls	Fused Volume and Flux with LUT-based Node Look-up (fl); Shared memory utilization for contributions and improved register allocation (s)	5.4.1, 5.4.2, and 5.4.3
GPU_fln	Fused Volume and Flux with LUT-based Node Look-up (fl); Node-tiling (n)	5.4.1 and 6.1
GPU_flsn	Fused Volume and Flux with LUT-based Node Look-up (fl); Shared memory utilization for contributions and improved register allocation (s); Node-tiling (n)	5.4.1, 5.4.2, 5.4.3, and 6.1
GPU_uln	Unified Volume, Flux, and Integration with LUT-based Node Look-up (ul); Node-tiling (n)	6.1 and 6.2
GPU_ulsn	Unified Volume, Flux, and Integration with LUT-based Node Look-up (ul); Shared memory utilization for contributions and improved register allocation (s); Node-tiling (n)	5.4.2, 5.4.3, 6.1, and 6.2
GPU_ulssn	Unified Volume, Flux, and Integration with LUT-based Node Look-up (ul); Shared memory utilization for contributions, shared memory utilization for variables, and improved register allocation (ss); Node-tiling (n)	5.4.2, 5.4.3, 6.1, and 6.2

Table 6.3: The GPU kernel configuration flavors for performance evaluation with different optimization levels and intra-device communication-reducing strategies.

The GPU_fln is a new flavor combining GPU_fl with the node-tiling strategy discussed in Section 6.1. Likewise, the GPU_flns combines GPU_fls with the node-tiling strategy. On the other hand, the GPU_uln replaces the fused volume and flux kernel on GPU_fln with the new unified volume, flux, and integration kernel discussed in Section 6.2. Finally, GPU_ulsn and GPU_ulssn add shared memory utilization and improved register allocation. Only contributions are stored in the shared memory for GPU_ulsn, while for GPU_ulssn, both contributions and variables are stored inside the shared memory, and hence, the double ss.

6.6.2 Simulation Runtime

This section discusses the effectiveness of the algorithms mentioned in Sections 6.1 and 6.2 to reduce the intra-device communications, by comparing it to the previous kernel optimization strategies discussed in Section 5.5.1. Figure 6.16 extends the Figure 5.6 by adding five new configurations (i.e., kernel sets) shown in Table 6.3. The execution time is measured for running the wave simulation for 1000 time steps. The simulation is run in 3D space at refinement level 5 with 32,768 elements on single NVIDIA Tesla V100 GPU. The new unified kernel is drawn in grey, representing the time needed to execute volume, flux, and integration in one kernel. While Figure 6.16 looks complicated with much data (i.e., eight kernel sets with six different wave simulation configurations), the following subsections group the kernel sets to make analysis easier.

6.6.2.1 `GPU_fl` vs. `GPU_fln`

This subsection compares the fused volume and flux kernel without and with the node-tiling. On average, the node-tiling improves the performance of the `GPU_fln` wave simulation kernel by 20% compared to `GPU_fl`. The Elastic-Central gets 37% and 27% performance improvements with node-tiling for FP64 and FP32, respectively. This is due to the improved element locality on the SM, reducing the intra-device communication between the SMs. However, Elastic-Riemann has the lowest performance benefit with node-tiling: 11% and 12% for FP64 and FP32, respectively. This is due to the greater number of intermediate results in the Riemann flux solver than the Central flux solver. The number of register spilling can be reduced by assigning more registers per thread and reducing the size of the tile from 128 nodes per tile to 64 nodes per tile. Finally, Acoustic gets 15% and 17% performance improvements with node-tiling for FP64 and FP32, respectively. The Acoustic kernel is significantly simpler and has a significantly less intermediate result. The Acoustic element data structure is also smaller than the Elastic element data structure (Table 4.1), making

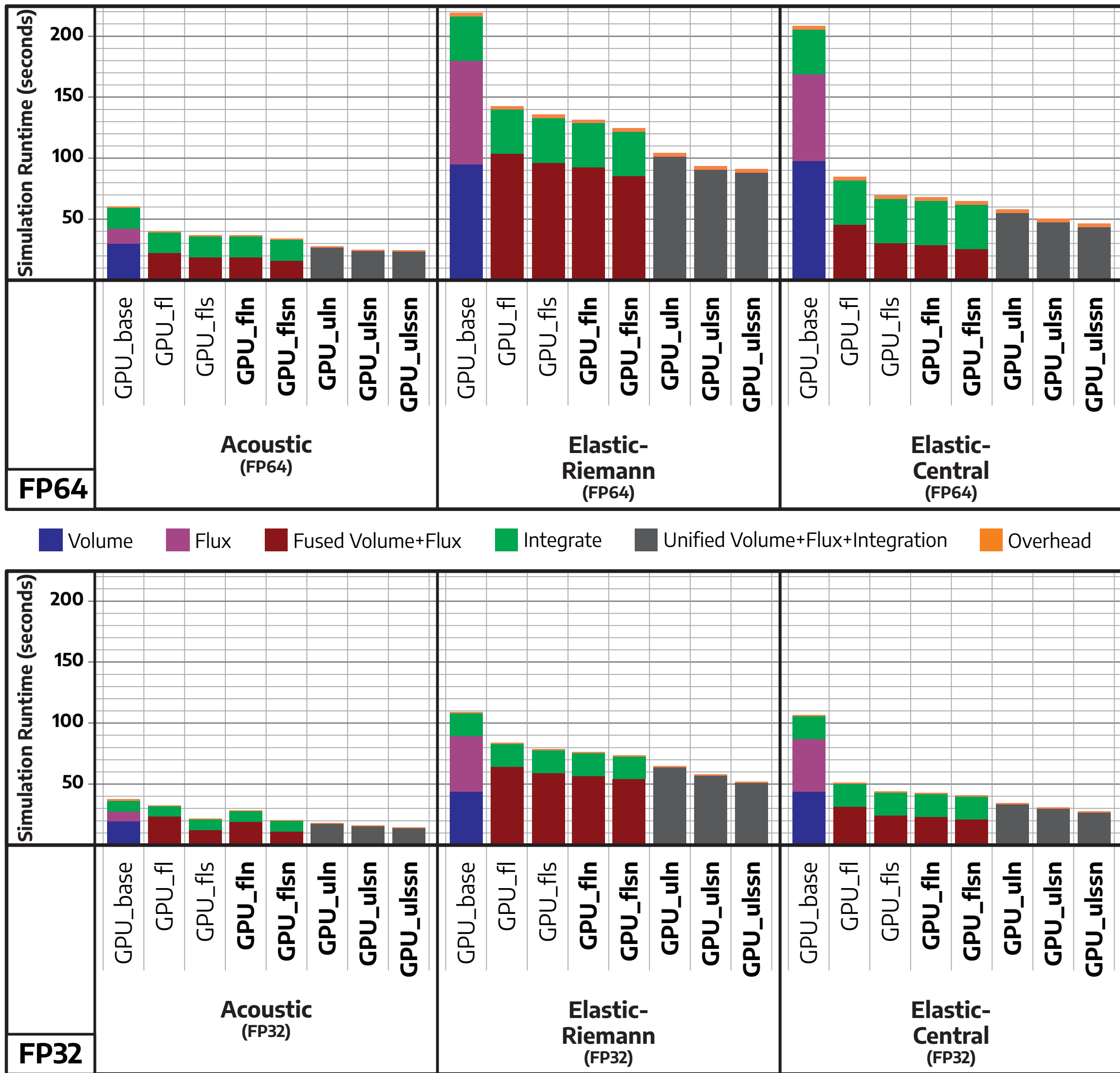


Figure 6.16: GPU simulation time and the total execution time for each kernel for 3D space at refinement level 5 (32,768 elements) running on single NVIDIA Tesla V100 GPU for 1000 time steps. This figure extends Figure 5.6 by adding five new kernel sets that apply communication-reducing strategies discussed in Chapter 6. The basic kernel (Section 5.3) uses separate volume (dark blue) and flux (purple) kernels, while the optimized kernel (Section 5.4) uses fused volume and flux kernels (dark-red). The unified kernel (grey) merges volume, flux, and integration into single kernel. Except for the unified kernel, the integration (green) kernel is the same across different flavors. The overhead is the additional time for preparing the simulation (e.g., constructing LUT, copying mesh data from CPU to GPU memory) and finishing the simulation (e.g., copying back the mesh data from GPU to CPU memory).

it easier to fit inside an SM, even without the node-tiling. Since it already enjoys higher performance due to the better element locality, the Acoustic kernel sees lower performance improvements with node-tiling than the Elastic kernel.

6.6.2.2 `GPU_fls` vs. `GPU_flsn`

This subsection compares the fused volume and flux kernel with optimized register allocation and shared memory utilization without and with the node-tiling. The shared memory only stores the `contributions`. On average, the node-tiling improves the performance of the `GPU_flsn` wave simulation kernel by 13% over the `GPU_fls`. This improvement is lower than the one discussed in Section 6.6.2.1 since the `GPU_fls` already enjoy the benefit of improved register allocation and shared memory utilization, allowing the GPU to hide the memory access latency better since more thread blocks can be scheduled into one SM. The rest of the performance improvements come from the reduced intra-device traffic thanks to element locality. Acoustic and Elastic-Central enjoy 16% and 13% performance improvements for FP64 and FP32, respectively. The Elastic-Riemann remains the kernel that gets the least benefits from the node-tiling, only 10% and 8% for FP64 and FP32, respectively. Reducing the tile size (i.e., the number of nodes per tile) and the number of threads per thread block may be helpful for Elastic-Riemann in better managing the register usage for all threads.

6.6.2.3 `GPU_fln` vs. `GPU_flsn`

This subsection compares the effect of shared memory utilization and improved register allocation for fused volume and flux kernel with node-tiling. Storing the `contributions` inside the shared memory and improved register allocation can bring an average of 15% improvements to the kernel that already uses node-tiling. However, the average is higher since `GPU_flsn` in Acoustic with FP32 enjoys 43% improvements over `GPU_fln` thanks to the smallest element data size compared to others, allowing it

to stay inside the SM easily. The reduced L1 cache due to shared memory utilization has minimal impact on the performance of Acoustic with FP32. Removing this outlier, the average performance improvements of `GPU_flsn` over `GPU_fln` is 10%.

6.6.2.4 `GPU_fln` vs. `GPU_uln`

This subsection compares the total runtime for the simulations (volume, flux, and integration) when using fused volume and flux kernel with a separate integration kernel and the unified kernel. Here, both kernel sets, `GPU_fln` and `GPU_uln`, use node-tiling for apple-to-apple comparisons. On average, using the unified kernel brings 22% overall performance improvements over fusing the volume and flux only, thanks to the reduced data movement and the overhead of launching integration as a separate kernel. Again, Acoustic with FP32 enjoys the most benefit of using the unified kernel, with 37% improvement over the fused volume and flux kernel. On the other hand, the Acoustic with FP64 enjoys 25% performance improvements. For Elastic-Central, the performance improvements are 15% and 19% for FP64 and FP32, respectively. Finally, for Elastic-Riemann, the performance improvements are 21% and 15% for FP64 and FP32, respectively.

6.6.2.5 `GPU_uln` vs. `GPU_ulsn`

This subsection compares the effect of having shared memory for storing the `contributions` and improved register allocation to the unified kernel. On average, the performance improvement is 11%, with Elastic-Central enjoying the most benefit with 13% improvement.

6.6.2.6 `GPU_ulsn` vs. `GPU_ulssn`

This subsection compares the effect of using shared memory to store both `variables` and `contributions` on the unified kernel. Previously in Section 5.4.2, only the `contributions` is stored in shared memory since, without the node-tiling,

the element is split into multiple thread block, reducing the effectiveness of storing the `variables` inside shared memory. This time, with node-tiling and unified kernel, storing both `variables` and `contributions` brings an average of 7% performance improvements. The Acoustic with FP64 gets only 1% performance improvements while the Elastic-Riemann with FP32 gets 11% performance improvements.

6.6.3 Key Takeaways

The node-tiling brings an average of 20% performance improvements with up to 37% higher performance compared to the kernel without node-tiling. Using shared memory and improved register allocation gives another 15% average performance improvements on the kernel with node-tiling. The Elastic-Riemann may benefit from smaller tile size due to the amount of the intermediate results, requiring more registers per thread. The unified kernel merging volume, flux, and integration into one kernel significantly improves the performance by an average of 22% compared to the fused volume and flux kernel with a separate integration kernel. Using shared memory for storing `contributions` and improved register allocations bring additional performance improvement of 11% on average. Storing both `contributions` and `variables` inside the shared memory brings additional 7% average performance improvements. Finally, node-tiling and the unified kernel are effective in reducing intra-device communication overhead, which can be combined with optimization techniques described in Section 5.4 to enhance the performance of the wave simulations. However, each GPU architecture and each workload may require fine-tuning (e.g., tile size, register per thread) to yield the highest performance possible.

6.7 Inter-Device Communication Performance Evaluation

This section evaluates the inter-device communication-reducing strategies discussed in Sections 6.3 and 6.4: the face-nodes-only ghost exchange and the reduced precision ghost exchange. Both strategies aim to reduce the amount of data trans-

ferred during the ghost exchange. Unlike the optimization method described in Section 5.6.2 that virtually costs nothing to implement, the face-nodes-only and reduced-precision ghost exchange strategies may cost additional computation and loss of numerical accuracy, respectively. Finally, the partial ghost exchange, discussed in Section 6.5, is still in early exploration and has not been made into GPU implementation. Thus, it is not evaluated in this section.

6.7.1 Evaluating Face-Node-Only Ghost Exchange

This section evaluates the Face-Nodes-Only ghost exchange strategy, discussed in Section 6.3. First, the inter-device communication overhead is compared for intra-node and inter-node communications. Then, the end-to-end wave simulation performance is evaluated to identify additional computation overhead associated with this strategy. The TACC Longhorn cluster (Section 3.2.2) is used for the evaluation performed in this section. Finally, the early performance evaluation on a newer platform, the DGX-A100, is discussed, and this strategy is employed to reduce inter-device intra-node communication overhead.

6.7.1.1 Comparing Ghost Exchange Overhead

Table 6.1 shows the size comparison between the standard `ElementDataBase`, the `ElementDataBaseGhostV1`, and the `ElementDataBaseGhostV2`. The latter has 42% smaller size than its predecessor and is 84% smaller than the standard element size. Figure 6.17 shows how this smaller size translates to the real-world performance across six different configurations of wave simulations. In single node with four NVIDIA Tesla V100 GPUs handling 262 thousand elements, the Face-Nodes-Only ghost exchange reduces the communication overhead by an average of 81% compared to the baseline implementation using `ElementDataBase`. Compared to its predecessor, which uses `ElementDataBaseGhostV1`, it reduces the intra-node communication overhead by 39%.

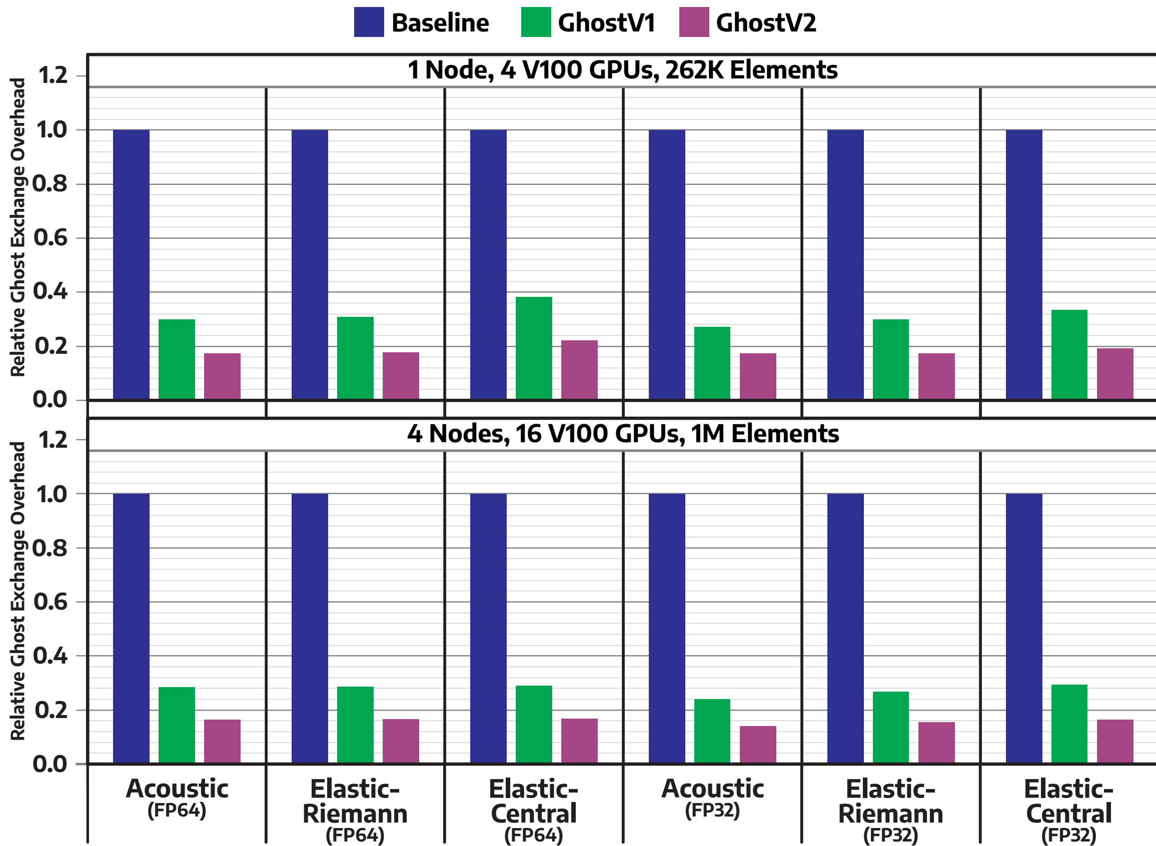


Figure 6.17: The improvement of multi-GPU simulation performance after reducing the size of ghost elements being exchanged by 42% over predecessor and 84% over the default size, as shown in Table 6.1. This figure is an extension of Figure 5.14. On average, the simulation time is reduced by 83% compared to the baseline, where multi-node multi-GPU runs benefit the most. Note that Baseline refers to `ElementDataBase`, GhostV1 refers to `ElementDataBaseGhostV1`, and GhostV2 refers to `ElementDataBaseGhostV2`.

In addition, for four nodes with a total of sixteen NVIDIA Tesla V100 GPUs handling 1 million elements, Face-Node-Only ghost exchange reduces the inter-node communication overhead by an average of 84% compared to the baseline implementation using `ElementDataBase`. Compared to its predecessor, it reduces the inter-node communication overhead by 42%. In other words, the reduction of the volume of data transferred during the ghost exchange is almost linear to the reduction of communication overhead for intra-node and inter-node.

6.7.1.2 Additional Compute Overhead

While the data volume reduction yields favorable results in reducing the intra-device communication overhead, the question remains whether Face-Node-Only ghost exchange introduces additional compute overhead when running the external flux kernel. This extra overhead calculates the node index mapping to the `variables` array using the Algorithm 6, from the regular index to the face-node-only index.

Figure 6.18 shows end-to-end evaluation for Face-Node-Only ghost exchange compared to the baseline and its predecessor for intra-node and inter-node communications. The `GPU_fls` is used as the kernel, and only the external flux is responsible for computing the node index mapping. Although only Elastic-Central and Elastic-Riemann with FP64 are shown, the other four configurations show similar patterns. Single-node with four GPUs and dual-node with eight GPUs handle the same amount of 262 thousand elements running for 1000 time steps.

While the ghost exchange overhead (red) is significantly reduced following the pattern shown in Figure 6.17, the time spent for computing the external flux is almost the same for the three scenarios of the ghost exchange. In single-node and dual-node runs, the external flux in Face-Node-Only ghost exchange consumes less than 1% more times than the baseline and its predecessor on average. Considering the external flux is significantly smaller than the volume and internal flux combined with the integration, this additional computation overhead is negligible. This is also the

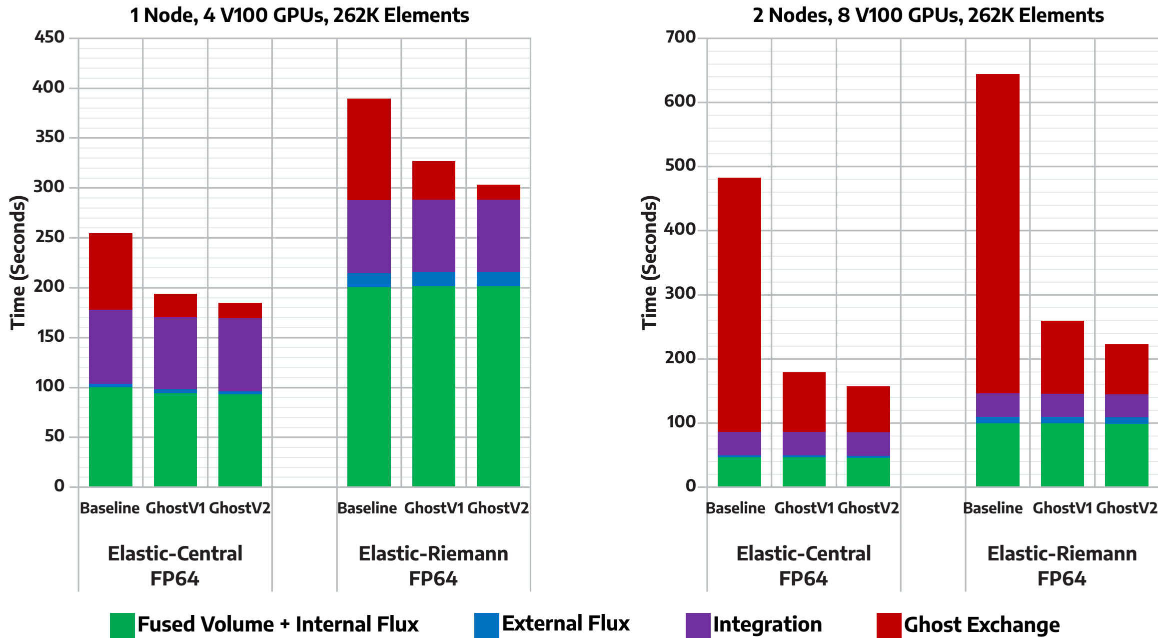


Figure 6.18: The end-to-end wave simulations performance on two compute nodes for a total of eight GPUs, running Elastic-Central and Elastic-Riemann in FP64 with 262 thousand elements for 1000 time steps. The external flux, which has the burden of computing the node index mapping, is shown to have a minimal runtime impact with less than 1% of additional time observed. The other four configurations of wave simulations follow the same patterns. Note that Baseline refers to `ElementDataBase`, GhostV1 refers to `ElementDataBaseGhostV1`, and GhostV2 refers to `ElementDataBaseGhostV2`.

reason why using LUT to store the node index mapping, discussed in Section 6.3.3.2, is not explored since the Algorithm 6 already yields satisfactory result. It is also worth mentioning that Algorithm 6 comprises only integer operations executed by the less-busy integer units inside the SM of the GPU.

6.7.1.3 Early Performance Exploration on Newer Platform

While previous evaluation uses the TACC Longhorn cluster (Section 3.2.2) to perform the experiments, this section serves as an early performance exploration of a new computing platform, the DGX-A100, as discussed in Section 3.2.4. The DGX-A100, described in NVIDIA Corporation (2020b), is a purpose-built platform

for handling state-of-the-art machine learning workloads. It ensures that the interconnection between the GPUs inside the node and between GPUs from other nodes has the highest bandwidth as the technology at that time allows. All eight NVIDIA A100 GPUs inside DGX-A100 are connected using NVLink 3.0, providing 600 GBps bidirectional bandwidth between GPUs. As a note, the inter-device intra-node bandwidth is 0.67×the HBM2 bandwidth in NVIDIA V100 in TACC Longhorn.

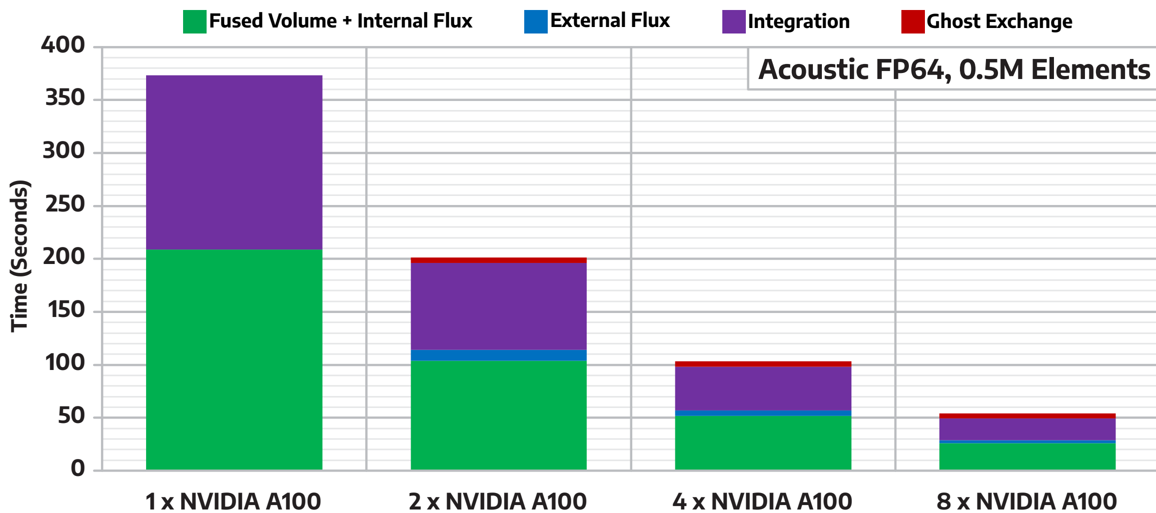


Figure 6.19: The performance evaluation of multi-GPU wave simulation on a newer computing platform, the DGX-A100, comprising eight NVIDIA A100 GPUs handling half a million elements for 1000 time steps. It uses `GPU_fls` kernel and face-node-only ghost exchange. Although it is only shown for Acoustic with FP64, the other five configurations of wave simulations follow a similar pattern. With the sheer amount of bandwidth between the GPUs and the reduced ghost exchange overhead, the wave simulations can achieve near-perfect strong scaling across two, four, and eight GPUs.

Figure 6.19 shows the early multi-GPU wave-simulation performance evaluation using DGX-A100, running `GPU_fls` kernel on Acoustic with FP64 for 0.5 million Elements. The Face-Node-Only ghost exchange is used to reduce the ghost exchange overhead. The MVAPICH2-GDR is used as an MPI library and is compiled from scratch with `ch3:mrail` and with `MV2_USE_GPUDIRECT_LOOPBACK` disabled. With the sheer amount of bandwidth between the GPUs and the significantly reduced ghost exchange overhead, the wave simulation can achieve near-perfect strong scaling across

two, four, and eight NVIDIA A100 GPUs. The other five configurations of wave simulations follow similar performance patterns.

In addition, each NVIDIA A100 GPU has 40 GB of HBM2 memory, 2.5×larger than the NVIDIA Tesla V100 GPU. This means it can fit 2.5×more elements in each GPU, making the GPU more busy processing the local elements in its sub-mesh instead of performing more ghost exchanges. The larger portion of local computation also helps with more potential overlapping between computation and communication when MPI asynchronous progression (Section 5.6.3) is enabled.

6.7.2 Evaluating Reduced-Precision Ghost Exchange

This section evaluates the Reduced-Precision ghost exchange strategy, discussed in Section 6.4. First, the inter-device communication overhead is compared for intra-node and inter-node communications, using both ghost element data structures introduced in Sections 5.6.2 and 6.3 for various data precision, including double-, single-, and half-precision floating-point data types, as shown in Table 6.2. Since reducing the precision will potentially impact the numerical accuracy, the end-to-end wave simulation performance is evaluated across two compute nodes for various ghost exchange precision to compare the L_2 error (Section 4.6.1).

6.7.2.1 Comparing Ghost Exchange Overhead

Figure 6.20 shows the ghost exchange overhead comparison when reduced-precision ghost elements are used, extending the results shown in Figure 6.17. Going from double-precision to single-precision reduces the ghost element size by half, and going further to half-precision reduces it by another half. As observed in the previous experiment discussed in Sections 5.6.2 and 6.7.1.1, the reduction of ghost exchange overhead is proportional to the reduction of the volume of data transferred.

In single-node run with four NVIDIA V100 GPUs handling 262 thousand elements, the use of single-precision and half-precision reduce the ghost exchange

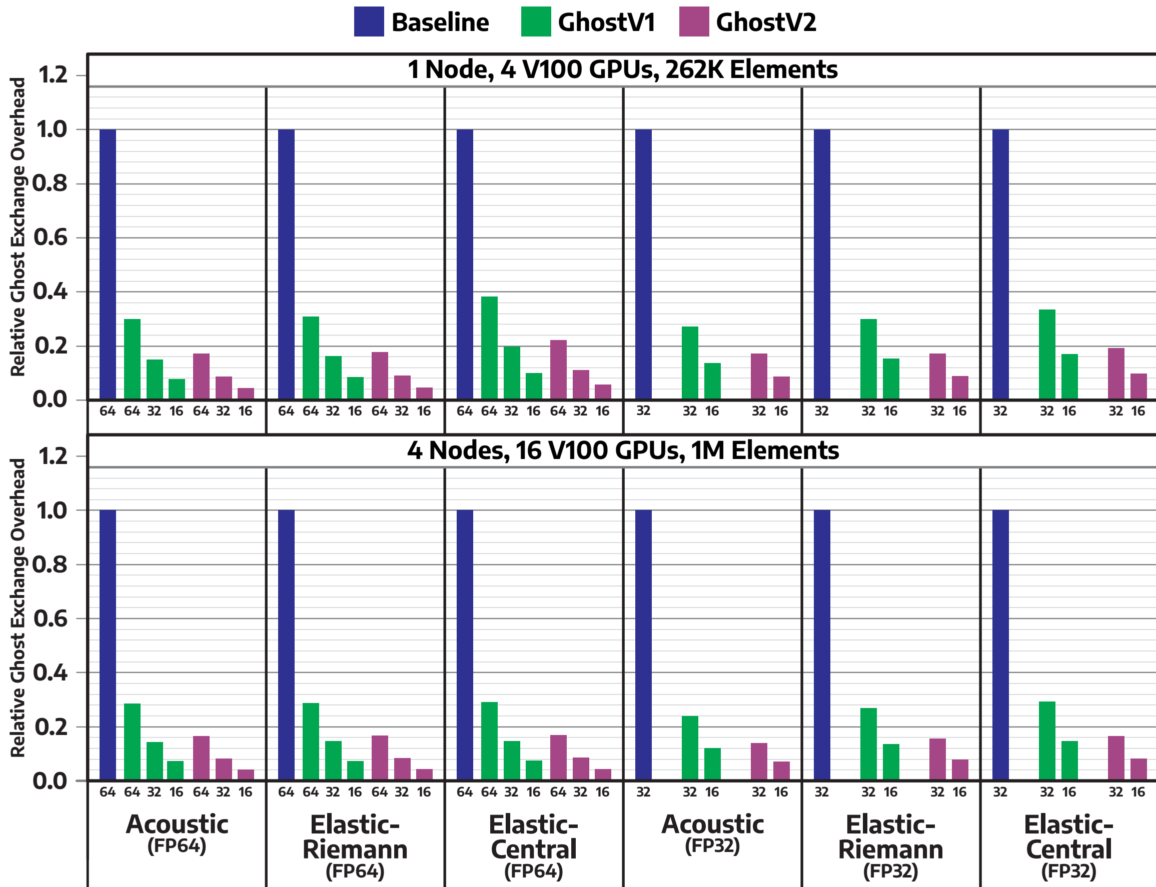


Figure 6.20: The improvement of multi-GPU simulation performance after reducing the size of ghost elements being exchanged by 50% and 75% through the use of lower precision data types: single- and half-precisions, as shown in Table 6.2. On average, the simulation time is reduced by 49.17% and 74% when going from double-precision to single- and half-precision for ghost exchanges. Note that Baseline refers to `ElementDataBase`, GhostV1 refers to `ElementDataBaseGhostV1`, and GhostV2 refers to `ElementDataBaseGhostV2`.

overhead by an average of 48.6% and 73.2% for `ElementDataBaseGhostV1` when FP64 is used as primary precision to run the wave simulation, respectively. When FP32 is used as primary precision, the half-precision ghost element reduces the ghost exchange overhead by an average of 48.9%. Note that when using FP32 as primary precision, double-precision ghost elements do not make any sense; thus, it is not considered. For `ElementDataBaseGhostV2`, the pattern is similar: 49.56% and 74.2% average reduction in ghost exchange overhead when using single-precision and half-precision, respectively, for ghost exchange with FP64 primary precision and 49.31% average reduction in ghost exchange overhead when using half-precision for ghost exchange with FP32 primary precision.

Similar patterns are observed with dual-node run with two compute nodes for a total of eight NVIDIA V100 GPUs handling one million elements. Using single-precision and half-precision for `ElementDataBaseGhostV1` reduce the ghost exchange overhead by an average of 49% and 74.2%, respectively, with FP64 as primary precision. For FP32 as primary precision, using half-precision reduces the ghost exchange overhead by an average of 49.2%. Likewise, for `ElementDataBaseGhostV2` with FP64 as primary precision, using single-precision and half-precision reduce ghost exchange overhead by an average of 49.4% and 74.4%, respectively. With FP32 as primary precision, the use of half-precision reduces ghost exchange overhead by an average of 49.17%.

6.7.2.2 Impact on Numerical Accuracy

While it has been shown that reducing the precision of the data being transferred during the ghost exchange can significantly reduce the overhead of ghost exchange, a critical question to answer is how much numerical accuracy is sacrificed. To investigate the numerical accuracy, end-to-end performance evaluation of the wave simulations are performed across two compute nodes for a total of eight NVIDIA V100 GPUs handling 262 thousand elements for 1000 time steps. The precision combinations follow the strategy in Figure 6.13. The performance of the wave simulations

for Acoustic and Elastic-Central as primary precision with `GPU_fls` kernel and Face-Node-Only ghost exchange is illustrated in Figure 6.21. The Elastic-Riemann follows similar patterns.

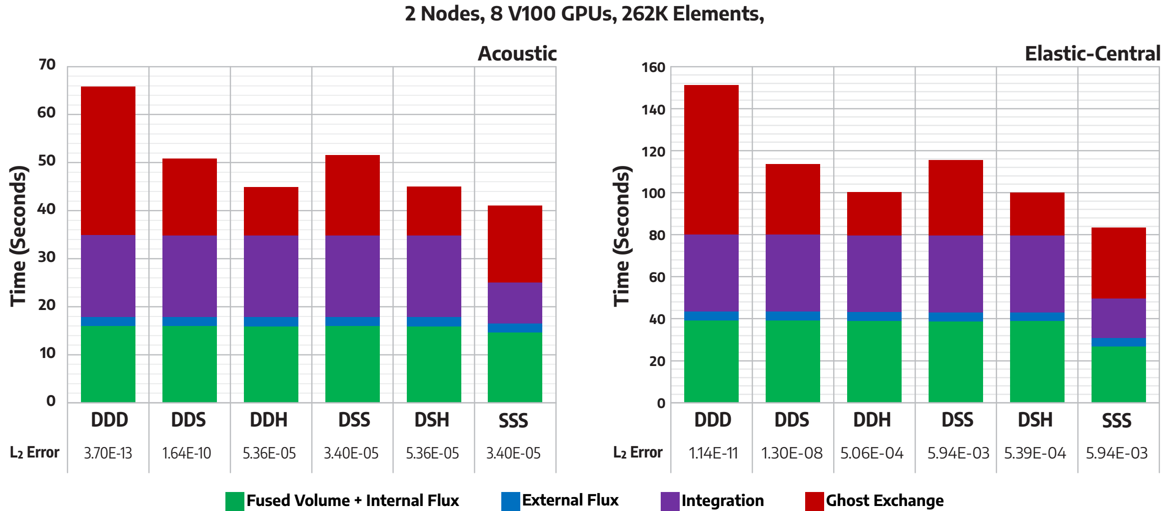


Figure 6.21: The comparison of wave simulation performance and numerical accuracy when using reduced-precision ghost exchange with strategies given in Figure 6.13. The wave simulations are run on two nodes for eight NVIDIA V100 GPUs handling 262 thousand elements for 1000 time steps. Using reduced precision ghost exchange generally results in a higher L_2 error. However, the runs are completed without any numerical instability.

The `DDD` and `SSS` are the regular one-precision ghost exchanges as reference. For the Acoustic, the L_2 errors are 3.70×10^{-13} and 3.40×10^{-5}, respectively, while for the Elastic-Central, the L_2 errors are 1.14×10^{-11} and 1.14×10^{-3}, respectively. Reducing to single-precision ghost element while keeping double precision to compute external flux (i.e., `DDS`) increases the L_2 error to 1.64×10^{-10} and 1.30×10^{-8} for Acoustic and Elastic-Central, respectively. Further reduction to half-precision ghost element increases the L_2 error to 5.36×10^{-5} and 5.06×10^{-4} for Acoustic and Elastic-Central, respectively. However, this L_2 error is similar to `SSS` while only transmitting half the volume of data during the ghost exchange. Note that `SSS` gains the advantage in computation since single-precision arithmetic is faster than double-precision arithmetic. It is worth noting that all configurations of wave simulations in

this experiment complete their execution without numerical instability issues.

With the precision reduction on ghost elements, one may question whether computing the external flux in reduced precision is also beneficial while not affecting the numerical accuracy too much. The `DSS` and `DSH` answer this question, with L_2 error in the same range as the `SSS`. Using single-precision for computing external flux is an advantage as the hardware can execute single-precision arithmetic faster than double-precision.

6.7.3 Key Takeaways

The Face-Node-Only ghost exchange brings an average of 82% and 41% lower inter-device communication overhead against the baseline using `ElementDataBase` for ghost exchange and its predecessor using `ElementDataBaseGhostV1` for ghost exchange, respectively. Interestingly, the node index mapping that the external flux must calculate gives negligible additional compute overhead (i.e., less than 1%), and thus, the use of LUT for node index mapping is not explored. Although the Face-Node-Only ghost exchange does not affect the numerical accuracy, another strategy called Reduced-Precision ghost exchange impacts the numerical accuracy, measured using L_2 error (Section 4.6.1). The reduced-precision ghost exchange can significantly reduce the size of ghost elements by 50% and 75% when going from double-precision to single- and half-precision, respectively. This translates to 74.06% and 49.17% of the average reduction in ghost exchange overhead. Although it affects the numerical accuracy with higher L_2 errors as a lower precision element is used for ghost exchange, all wave simulations performed during the experiment complete their run without any numerical instability issues. Finally, although it is still in the early exploration stage, the partial ghost exchange is worth investigating as it promises to reduce the volume of data transmitted during the ghost exchange at the expense of extra computation for approximating missing `variables`. Nevertheless, it must be proven to yield a stable formulation when running on CPUs first before developing the GPU-accelerated codes.

Chapter 7: PIM Architecture for Accelerating dG-based Wave Simulations

This chapter[1] describes the third contribution of this dissertation: accelerating the dG-based wave simulations using Processing-in-Memory (PIM), an emerging computing technology. The work done in this chapter has been published as a paper with the title *Wave-PIM: Accelerating Wave Simulation Using Processing-in-Memory* by Hanindhito et al. (2021).

As described in Chapters 5 and 6, GPU is a promising candidate for accelerating wave simulations. The work by Chan et al. (2016) describes that the GPU can provide a massive number of parallel executions, which is suitable for extracting the inherent parallelism available in wave simulations. However, the amount of data to process significantly exceeds the capacity of the on-chip memory of modern GPUs. The overhead of the data movement between on-chip and off-chip memory quickly becomes the new bottleneck for the GPU (Section 5.5.3), even though dG has a lot of data locality to the computation (Section 2.3.1) and efforts have been done to reduce the data movement overhead (Section 5.4). This phenomenon is called the memory wall, as described by Wulf and McKee (1995), and has become the key performance bottleneck of modern hardware accelerators, including GPUs, as described by Imani et al. (2019a, 2020).

In addition to the limited on-chip memory, the data movement introduced by the inter-element data synchronization during the flux computation is a key fac-

[1]Contents of this chapter have been published and appear in:
Bagus Hanindhito, Ruihao Li, Dimitrios Gourounas, Arash Fathi, Karan Govil, Dimitar Trenev, Andreas Gerstlauer, and Lizy K. John. 2021. Wave-PIM: Accelerating Wave Simulation Using Processing-in-Memory. the 50th International Conference on Parallel Processing (Virtual Event) (ICPP '21). Association for Computing Machinery, New York, NY, USA, Article 8, 11 pages. https://doi.org/10.1145/3472456.3472512.
I proposed, designed, implemented, and evaluated the main ideas while collaborating with the co-authors on writing the manuscript.

tor affecting the performance of the GPUs. Furthermore, GPUs have high energy consumption, which has become a key concern for data centers and supercomputer clusters. Therefore, finding new, more efficient massively parallel hardware architecture with high performance and low energy consumption for wave simulation is highly attractive.

Processing-in-memory (PIM) is an emerging computing technology that attempts to address the bottleneck between the compute units and the off-chip memory in von Neumann architectures, improving performance and reducing energy consumption. PIM technology allows the computation to be performed inside the memory arrays where the data is stored, eliminating the need for data movement between compute units and off-chip memory. PIM has been briefly discussed in Section 2.7. This chapter develops PIM architecture utilizing emerging resistive memory technology, which has excellent potential to become an in-memory accelerator since it can perform parallel computation directly inside the crossbar logic, as described by Talati et al. (2016); Haj-Ali et al. (2018).

First, the PIM architecture is described. It is designed to accelerate wave simulation (Section 7.1). Since it is an emerging technology, the availability of robust software libraries or frameworks is limited to none, and thus, all of the work to port the CPU code to target PIM is done manually. Two types of interconnection between the memory blocks are described, and their advantages and disadvantages are identified. Additionally, implementing a Look-up Table on PIM for storing the mesh structure, like in GPU code (Section 5.1.2.2), is explained.

After taking care of the PIM architecture, the basic implementation of wave simulation kernels that can run on it are described (Section 7.2). This includes storing the data inside the resistive memory arrays with a layout that is optimized for parallel execution and the execution timeline, which takes into account the capability of the memory blocks. The data mapping and layout techniques explored here can also be applied to other SRAM- and DRAM-based processing-in-memory technologies, as

described by Eckert et al. (2018); Gao et al. (2019).

Next, optimization efforts were performed to improve the PIM architecture's scalability, performance, and energy efficiency for handling the wave simulation (Section 7.3). This includes batching techniques to support larger problem sizes, folding techniques to improve parallelism of smaller problem sizes, and pipelining to improve the throughput. Finally, the performance and energy efficiency of wave simulation running on PIM are compared against GPU (Section 7.4). I also compare the two interconnection topologies between memory blocks, the pipelining techniques, and the numerical accuracy losses. With minimal adjustments, the techniques and strategies developed here can be adopted in many other fields, such as in full-waveform inversion (e.g., works by Fathi et al. (2015b,a)), electromagnetic waves (e.g., work by Chen and Liu (2013)), and other hyperbolic partial differential equations (e.g., work by Quarteroni and Valli (1994)).

7.1 Architecture for Wave Simulation

This section introduces the PIM architecture used to accelerate wave simulations. It starts by briefly discussing the hierarchical organization of the PIM chips followed by comparing on-chip interconnection topologies. Since the software library and framework are not as robust as in GPU, manual efforts must be made to port the CPU code to run on the PIM hardware, which will be discussed next. Finally, it also explains how to integrate a look-up table (LUT) into PIM, a critical structure for representing the mesh structure, and the trade-off of two data types with which the computation in PIM is performed.

7.1.1 Hierarchical Organization

The proposed PIM architecture is a digital PIM (Section 2.7.2) constructed using memristors, a resistive memory technology. The organization of a PIM chip used in this dissertation is given in Figure 7.1. A PIM chip, shown in green, consists

of several memory tiles organized in a 2D array and a central controller. The number of memory tiles can be adjusted to get higher-capacity PIM chips. Some of the PIM chip characteristics used in this dissertation can be found in Table 3.4.

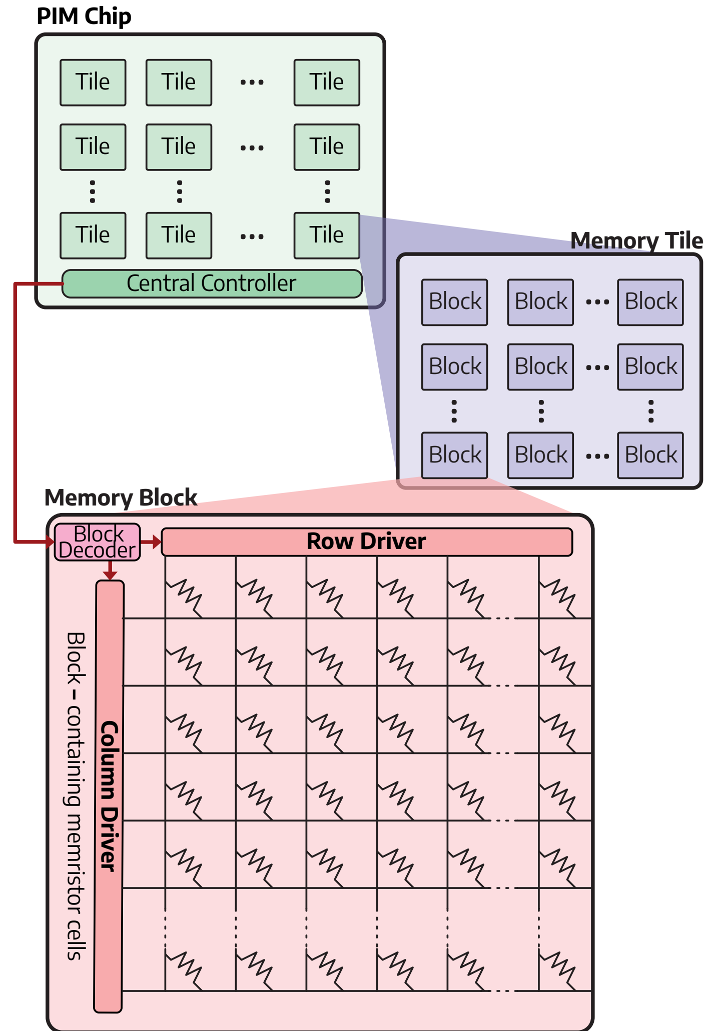


Figure 7.1: The architecture of PIM and its hierarchical organization. A PIM chip consists of a 2D array of memory tiles. Each tile comprises memory blocks, which are the smallest operation units of the PIM. Inside each memory block, memristor cells are organized in crossbar arrays.

Inside each memory tile are 256 memory blocks arranged in a 2D array (16×16). The memory blocks, shown in red, are the basic units where the computation is per-

formed. Each memory block is identical and contains memristor cells organized in 1024×1024 2D-array for a capacity of 1 Mb (megabit). There are also sense amplifiers, decoders, row drivers, column drivers, row buffers, and column buffers. This follows the same structure used in works by Imani et al. (2019a, 2020). The memory block receives instructions from the central controller. It performs the computations in a bit-serial way, utilizing many NOR operations to construct more complex arithmetic operations without additional or separate arithmetic-logic units (ALUs) hardware.

7.1.2 Central Controller

The PIM architecture used in this dissertation is an instruction-set architecture (ISA), where the instruction format is defined by Li et al. (2020b). The central controller issues instructions to the memory blocks, and the router interconnects between the memory blocks. Each memory block can perform independent operations (i.e., two memory blocks can perform different computations simultaneously). However, only one operation at a time can be performed on the memristor cross-bar arrays inside each memory block.

7.1.3 Memory Block Interconnection

Although previous work by Song et al. (2017); Imani et al. (2019a); Li et al. (2020b) shows performance improvement over GPU thanks to the removal of data movement between off-chip and on-chip memory, there is still concern with data movement inside the PIM. For example, in the work by Li et al. (2020b), the data movement contributes 95.09% to the overall execution time in GPU. While the PIM reduces this ratio to 64.5%, it is still considerably high.

7.1.3.1 Interconnect Challenges

The data movement inside the PIM chip is due to inter-memory-block data exchanges. Since the size of the non-volatile memory array (i.e., the capacity of

the memory block) is limited, as shown by Dong et al. (2012), this data movement is unavoidable. When arithmetic operations inside a memory block need operands stored in other blocks, they are transferred using the interconnection between memory blocks. This internal data movement from one memristor crossbar array to another affects the overall performance. Thus, kernels' dataflow kernels need to be handled efficiently, especially for those with many intermediate results and those that operate on the producer-consumer model, relying on data stored in other memory blocks.

Other works have investigated this issue, such as the work by Imani et al. (2019a), who proposed parallel data transfer between neighboring memory blocks to improve performance. However, they did not explore data exchange between non-neighboring memory blocks, making it not a general solution. In addition, works by Song et al. (2017); Li et al. (2020b) used global buffers for data routing and considered identical latency for data transfer between memory blocks, obtained from NVSim (Table 3.3). However, this global buffer design is inefficient for memory blocks located adjacent to others.

The communication latency depends on the organization of memory blocks inside the memory tile (i.e., by row and column) and the data path width. Higher throughput can be achieved by setting the data path width equal to the number of rows and columns of memory blocks. However, having a wider data path means more leakage power from the interconnect when it is not used for transmitting data. This is a particular concern when designing the interconnect for some edge devices that want to utilize PIM technology.

There is an opportunity to improve the efficiency of data exchange between memory blocks globally while maintaining the efficiency of data transfer between memory blocks adjacent to others. Leveraging the ISA-based PIM architecture discussed in Section 7.1.2, I proposed an optimized H-Tree and Bus interconnect with flow controls to make the data transfer between memory blocks more efficient. The flow control is achieved by adding additional circuitry to implement routing switches.

7.1.3.2 H-Tree Interconnect

As shown in works by Leiserson (1980); DeWitt et al. (1984); Dong et al. (2012), H-Tree is a data structure that is widely used in many fields, including VLSI, database systems and DRAM interconnects. I propose implementing H-Tree as the interconnect topology for connecting the memory blocks. Figure 7.2 shows an example of the interconnect topology using H-Tree to connect the memory blocks inside a tile.

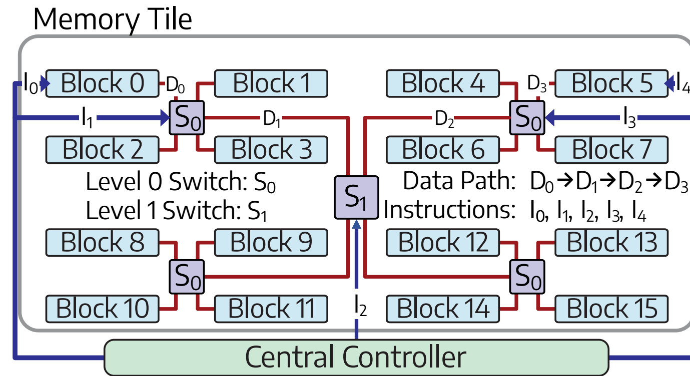


Figure 7.2: The topology of H-Tree interconnect for connecting memory blocks inside the memory tile. This example employs two-level routing switches, representing a tree structure. As long as the data transfers use different switches, they can run in parallel, improving communication efficiency between memory blocks.

In a tree structure, the memory blocks are the lowest level of the tree. Four memory blocks are connected using one routing switch, S_0. Then, a higher level routing switch (i.e., parent), S_1, connects four S_0 routing switches (i.e., children). Depending on the number of memory blocks and the number of children switches per parent; there can be multiple levels of routing switches. For example, with 256 memory blocks as used in this dissertation, there will be four levels of routing switches, if each switch can only have four children. If the parent switch has a small number of children, it requires more levels and routing switches, which consumes area and energy. However, many parallel data transfers can be performed if they do not use the same switches. The central controller issues instructions to each routing switch, depending on where the data needs to be transferred.

Using an example given in Figure 7.2, the data transfer between memory block 0 to memory block 5 using H-Tree topology can happen as follows. First, the central controller issues read instruction I_0 to block 0, asking it to load data from memristor cells into the row or column buffers. Then, the central controller issues instructions as a part of `memcpy` to the routing switches that control the path between memory blocks 0 and 5. That is, instruction I_1 for S_0 nearby the block 0, I_2 for S_1, and I_3 for S_0 nearby block 5. These instructions establish data path $D_0 \rightarrow D_1 \rightarrow D_2 \rightarrow D_3$. Finally, the central controller issues write instruction I_4 to block 5, asking it to store the data from the row or column buffers to the assigned memristor cells.

The H-Tree topology can lower data transfer latency between the memory blocks, compared to the previous PIM work by Li et al. (2020b). For example, consider an element with many nodes that cannot fit inside a memory block. In this case, multiple memory blocks are needed to store one element. Although, with dG, volume computation is a local operation to each element, splitting an element through multiple memory blocks incurs communication overhead between these memory blocks. With H-Tree topology and proper data layout (i.e., using memory blocks adjacent to each other to store an element data), the communication between these memory blocks is handled through the first level of the routing switch (S_0) without the need to go into a higher-level routing switch. In addition, other communications that only involve S_0 can happen in parallel, as long as they are not using the same S_0 switch.

7.1.3.3 Bus Interconnect

The bus interconnect is an alternative to the H-Tree interconnect, especially for smaller problems with little data transmission between memory blocks. For this type of problem, H-Tree may not be efficient due to the considerable leakage power consumed by the routing switches. For example, in a memory tile with 256 memory blocks used in this dissertation, $1+4+16+64=85$ routing switches need to be used in the H-Tree interconnect. Figure 7.3 shows an example of the interconnect topology using Bus to connect the memory blocks inside a tile.

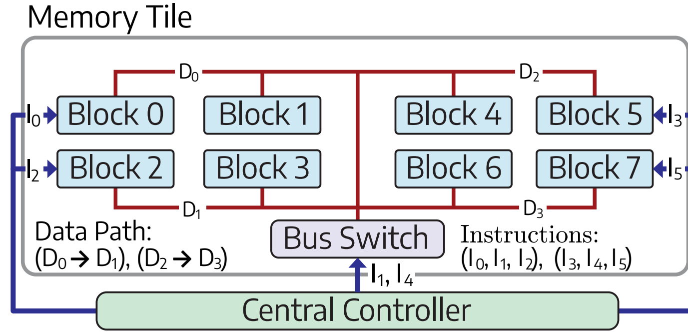


Figure 7.3: The topology of Bus interconnect for connecting memory blocks inside the memory tile. Unlike H-Tree, Bus interconnects only have one level of data path connecting all memory blocks within a tile. While it is simpler than H-Tree, only one data transfer can happen simultaneously.

Using an example given in Figure 7.3, two data transfers from memory blocks 0 to 2 and 5 to 7 happen as follows. First, the central controller issues two consecutive read instructions, I_0 and I_3, to block 0 and 5, asking both memory blocks to load the data from memristor cells into the row or column buffers. Then, the central controller issues I_1 instruction, instructing the bus switch to connect memory blocks 0 and 2 so data transfer between these blocks can be performed. After memory blocks 0 and 2 conclude their data transfer, the central controller issues I_4 instruction to the bus switch, asking it to connect memory blocks 5 and 7, allowing them to exchange data. Finally, the central controller issues two consecutive write instructions, I_2 and I_5, to blocks 2 and 7, asking them to store the data from the row or column buffers to memristor cells.

These two data transfers cannot happen simultaneously, unlike the H-tree interconnect. The Bus interconnect is less efficient for large problem sizes with intensive data transfer between memory blocks than the H-Tree. The performance comparison between H-Tree and Bus interconnect will be discussed in Section 7.4.5.

7.1.4 Look-up Table Implementation on PIM

Look-up Tables (LUTs) are commonly used in GPUs, FPGAs, and ASICs (e.g., as shown by Vinyals and Friedland (2008)). They are also used in high-performance computing applications, as demonstrated by Lucente (1993); Groeneveld et al. (1996). The GPU implementation of wave simulation, described in Chapter 5, uses many look-up tables, including storing the mesh structure, allowing it to find neighboring elements efficiently. LUTs allow complicated computations to be replaced with simpler array-indexing operations.

The `p4est` does not have built-in support for GPUs, let alone PIM, and thus I have to do the same for PIM code: storing the mesh structure in LUTs, just like what I implemented in GPU code (Section 5.1.2.2). Without LUTs, finding neighboring elements is impossible to do efficiently using just NOR operations since it will have a very long latency. However, using LUTs in PIM is not as straightforward as performing memory allocation, unlike the GPU. Some architectural features must be implemented before LUTs can be "allocated" and used in PIM.

The approach to implementing LUTs inside the PIM is to use dedicated ordinary memory blocks, instead of adding custom hardware units. The contents of the LUTs are loaded to these reserved memory blocks during the initialization phase of the simulations, just like in GPU code. Then, memory blocks can generate indexes for accessing the LUTs during the computation. This means that look-up table operations can be viewed as a special case of data transfer between memory blocks (i.e., the reserved block for LUT and the block requesting the data). Since it is an ISA-based PIM defined in Section 7.1.2), the look-up table operations can be encoded as PIM instructions.

Figure 7.4 shows the proposed instruction format for PIM based on ISA defined in prior work by Li et al. (2020b). The `Opcode` (bit 63rd to 57th) distinguishes the LUT instruction from other PIM instructions. The LUT index's memory address is calculated by shifting and adding the `Row ID` and the `Offset_S`. Assuming the

63–57	56–31	30–26	25–5	4–0
Opcode	Row ID	Offset_S	LUT Block ID	Offset_D

Figure 7.4: The instruction format added to the ISA for interacting with the Look-up Tables in PIM. The `Opcode` distinguishes this instruction with other PIM instructions.

Algorithm 7: Steps for executing one look-up table instruction.

Defined:

`Opcode` ▷ Distinguish instruction from other PIM instructions

Inputs:

`Row ID` ▷ Row ID containing LUT index

`Offset_S` ▷ Offset to calculate row address containing LUT index

`LUT Block ID` ▷ Block ID containing LUT contents

`Offset_D` ▷ Offset to calculate row address to store LUT contents

Outputs:

`R_1` ▷ Read instruction to read LUT index

`R_2` ▷ Read instruction to read LUT content

`W_1` ▷ Write instruction to write LUT content

1 **Generating** `R_1`**:**
Location to read LUT Index = `Row ID`×1024 + `Offset_S`×32
Size of LUT Index = 32 bits

2 **Issuing** `R_1` **to fetch 32-bit LUT index** (`idx`)

3 **Generating** `R_2`**:**
Location to read LUT Content = `LUT Block ID`×1024×1024 + `idx`×32
Size of LUT Content = 32 bits

4 **Issuing** `R_2` **to fetch 32-bit LUT content** (`content`)

5 **Generating** `W_1`**:**
Location to write LUT Content = `Row ID`×1024 + `Offset_D`×32
Size of LUT Content = 32 bits

6 **Issuing** `W_1` **to store 32-bit LUT content** (`content`)

memory block size is 1024×1024, and the compute precision is 32-bit, only 5 bits are needed for `Offset_S`. The address for the LUT content is calculated by adding the left-shift `LUT Block ID` and the LUT index. Finally, the content of the LUT is transferred to the destination address, defined by `Row ID` and `Offset_D`. Algorithm 7 shows the execution procedure of a single LUT instruction.

7.1.5 Interaction with Host CPU and Off-chip Memory

The host CPU sends instructions to the PIM chip and pre-processes parts of the input data. Complicated operations, such as square root and inverse, are "offloaded" from PIM to the host CPU since the latency will be too high for PIM to execute these operations using approximation techniques, such as Taylor series, as shown by Zhang et al. (2020a); Li et al. (2020b). The results of the operations will be stored in dedicated memory blocks by the host CPU, and the values can be obtained using the same procedure as LUT. The host CPU is assumed to have an architecture similar to ARM Cortex-A72, as described by Raspberry Pi Foundation (2021).

Furthermore, it is also assumed that the PIM chip has HBM2 off-chip memory with 900 GB/s bandwidth, a similar memory used by NVIDIA Tesla V100 GPU, as described by NVIDIA Corporation (2017). The power of this memory is 36.91 W, as shown by Li et al. (2018c).

7.1.6 Choice of The Data Types

There were prior works using fixed point data precision for machine learning, visualization, and other PIM applications, as shown in the works by Wang et al. (2019a); Li et al. (2020b); Chi et al. (2016). The accuracy loss of fixed point approximation is negligible in these applications. However, the accuracy loss of fixed point implementation cannot be ignored in the acoustic and elastic wave simulations. Hence, the PIM in this dissertation uses 32-bit floating-point data precision to perform the computation. However, it can also be extended to 64-bit at the expense of double

the arithmetic latency (Table 2.2). The floating-point operations are implemented with the search units proposed by Imani et al. (2019b).

Parameter	FP64	FP32	
	avg_ref	avg_diff (%)	max_diff (%)
Variables p	2.55E-01	1.09E-08 (0.000%)	1.08E-07 (0.000%)
Variables $\mathbf{v}$	1.28E-03	2.15E-09 (0.000%)	6.94E-08 (0.005%)
Contributions p	1.12E-08	5.56E-13 (0.005%)	1.01E-11 (0.090%)
Contributions $\mathbf{v}$	7.73E-07	1.05E-12 (0.000%)	3.60E-11 (0.005%)
Auxiliaries p	9.89E-06	3.90E-09 (0.039%)	1.58E-07 (1.594%)
Auxiliaries $\mathbf{v}$	3.50E-04	7.54E-09 (0.002%)	3.18E-07 (0.091%)
Average Percentage Error		0.008%	0.297%

Table 7.1: Accuracy comparison between double precision and single precision floating-point arithmetic in PIM

The single precision (32-bit floating-point) gives more promising performance improvement for PIM since the latency of the multiplication increases in proportion to the square of the data width. The comparison of accuracy loss from using single-precision floating point is given in Table 7.1. The overall average of accuracy loss is only 0.008% with a maximum loss of 0.297%. This number is acceptable considering PIM's shorter latency and performance improvement when using a 32-bit floating-point.

7.1.7 Mapping Wave Simulation to PIM

The wave simulations, as described in Chapter 4, can be decomposed into arithmetic instructions and general memory instructions. The host CPU sends these instructions to the PIM chip. The decoder inside the PIM chip central controller decodes these instructions. Then, the central controller generates and sends micro sequences to the memory blocks.

The host CPU performs the initialization, including the initialization of the `variables` based on the initial condition. The initialized data is then stored inside the PIM. The data layout inside the memory blocks within the PIM chip significantly affects the communication between memory blocks. Thus, it impacts the performance,

as discussed in Section 7.2.

When all required operands are ready, arithmetic instructions are issued. The computations are performed inside the memristor cells in a row-parallel way. This means each row inside a memory block performs the same calculations, and hence, there are up to 1024 row-parallel operations can be done in each memory block, assuming the memory block is 1024×1024. The results (outputs) of the operations are generated in the same row, and memory copy instructions are issued to store them. In addition to storing the final results, some columns in a row store the intermediate results, known as a scratchpad, as shown by Haj-Ali et al. (2018).

7.2 Basic Implementation of Wave Simulation

After establishing the architecture of the PIM for wave simulation, it is time to map the wave simulations described in Chapter 4 into PIM. There are two major steps for basic implementation: laying the data inside the memory blocks of the PIM and issuing instructions to perform the computation following the execution timeline. Both follow the CPU code's simulation flow and data flow, as described in Figures 4.9 and 4.10.

7.2.1 Data Layout in Memory

How the data is stored inside each memory block contributes to the amount of communication between memory blocks, thus impacting the performance of PIM. Following the 512-node elements used in this dissertation (Section 4.1.2) and the data structure used to represent each element (`ElementDataBase`, see Section 4.2.3), I discussed optimized data layout for both acoustic and elastic wave simulations to utilize 1024×1024 memory blocks of the PIM effectively. This data layout aims to reduce the communication between memory blocks and allows the memory blocks to perform as many row-parallel operations as possible.

7.2.1.1 Element Data Layout for Acoustic Wave Simulation

For acoustic wave simulation with four variables per node, the data layout in each memory block of PIM is illustrated in Figure 7.5. One element consisting of 512 nodes is stored inside one memory block. The first 512 rows of the memory block are used to store `variables`, `contributions`, `mass_inverse`, and `auxiliaries` where one node occupies one row. Storing these consumes 416 out of 1024 columns available per row since each data is 32-bit (i.e., single-precision floating-point). The rest of the columns are scratchpads to store intermediate results and broadcast constants. These 512 rows are where the computations are performed; hence, they are named computation rows. These 512 computation rows operate in parallel, allowing them to extract node-level parallelism (Section 4.1.4).

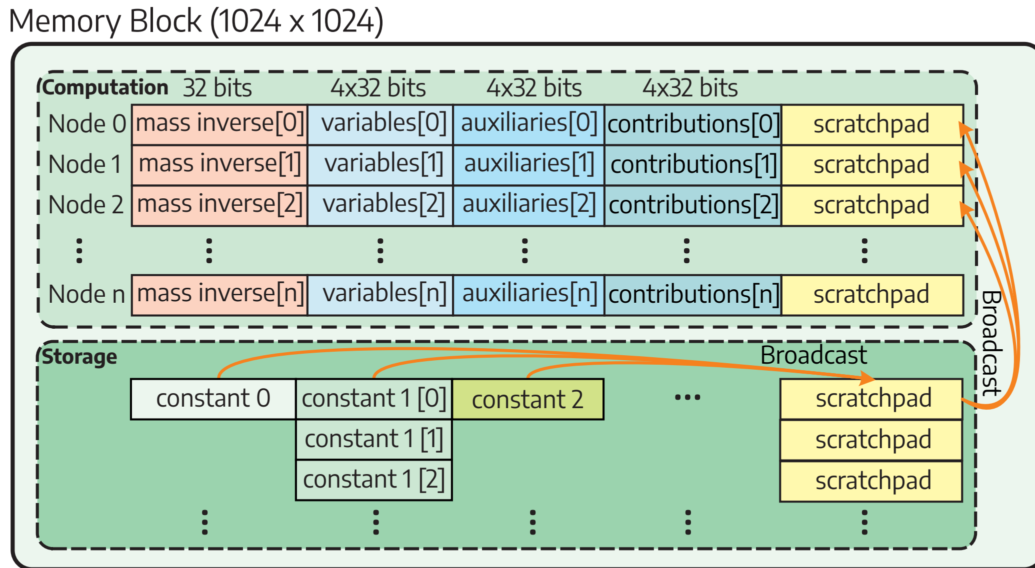


Figure 7.5: The memory data layout to store all members in `ElementDataBase` inside the memory blocks of PIM. Although it only illustrates the layout for acoustic wave simulation, elastic wave simulation follows a similar layout, except the nine variables must be split into multiple memory blocks due to insufficient capacity.

The last 512 rows are used to store constants; hence, they are named storage rows. These include the Gauss-Lobatto-Legendre (GLL) weights and integration points (Section 2.3.2 and Appendix B.4), precomputed derivative of shape functions

(i.e., similar to storing them in constant memory of GPU, see Section 5.3.2.1), and `materials`. These constants need to be copied to the scratchpad of each node by broadcasting them into the first 512 rows before row-parallel computation begins.

Since each memory block works independently, they can extract element-level parallelism (Section 4.1.4), especially for kernels that involve only local operations, such as volume and integration kernels. In refinement level 4 with 4096 elements (Section 4.1.1), there are 4,096 memory blocks used to store the elements (i.e., one memory block for one element). Likewise, the refinement level 5 with 32,768 elements needs 32,768 memory blocks.

7.2.1.2 Element Data Layout for Elastic Wave Simulation

Things get more complicated for elastic wave simulations with nine variables per node. The available 1024 columns per row are insufficient to store the data in each node, which requires 896 columns, leaving no sufficient room for scratchpad required for arithmetic operations, as described by Haj-Ali et al. (2018). Remember that, the elastic wave simulation kernels produce significantly more intermediate results, and thus, they need a bigger scratchpad. The number of columns is also limited by the circuit-level constraints, as shown by Dong et al. (2012). Therefore, each element must be split into multiple memory blocks using an optimization technique called expansion, discussed in Section 7.3.2.

7.2.2 Execution Timeline

After storing the elements' data using an optimized layout inside the memory blocks of PIM, it is now time to discuss how the computation is performed. That is, how the simulation kernels are translated into a sequence of PIM instructions, which are then executed in parallel by the memory blocks. Roughly, the execution timeline will follow the kernel execution flow detailed in Appendix B.4, with minor modifications, especially for elastic wave simulation kernels. Only volume and flux

kernels are discussed since the integration is straightforward.

7.2.2.1 Execution Timeline for Acoustic Wave Simulations

The top part of Figure 7.6 shows the execution timeline of the volume kernel for acoustic wave simulation. It resembles the volume kernel execution timeline in CPU, as shown in Figure B.9. After appropriate constants have been distributed to each row, the `jacobian_det_w_star`, the gradient of p (`grad_p`), and the divergent of $\mathbf{v}$ (`div_v`) are calculated by series of addition and multiplication. Both the `grad_p`[2] and `div_v`[3] are intermediate results and are stored in the scratchpad.

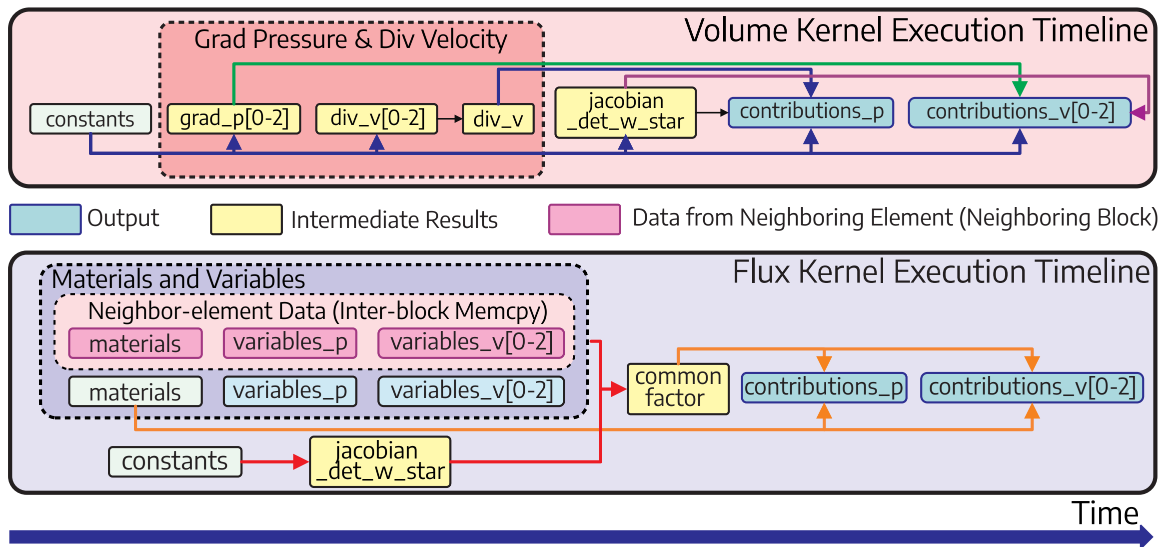


Figure 7.6: The execution timeline for volume kernel (top) and flux kernel (bottom). Although it only illustrates the execution timeline for acoustic wave simulation kernels, the elastic wave simulation kernels have a similar execution timeline, except there are additional data transfers from neighboring elements for both volume and flux kernels since the `variables` are split into multiple memory blocks.

Flux kernel has a similar execution timeline to the volume kernel, except for two significant challenges. The bottom part of Figure 7.6 shows the execution timeline

[2] i.e., the gradient of the pressure field, three-dimensional vector

[3] i.e., the divergent of particle velocity, scalar value

of the flux kernel for acoustic wave simulation. It resembles the flux kernel execution timeline in the CPU, as shown in Figure B.10, although, for simplification, the special case for boundary flux vector is not shown.

First, the flux kernel involves non-local operation, which requires obtaining `variables` from neighboring elements. This process demands data exchange between memory blocks, using either H-tree or Bus interconnect, as discussed in Section 7.1.3. Second, the flux kernel comprises several complicated arithmetic operations. For example, while computing the common factors and common values, there are square roots and inverse operations on `materials`. Since the `materials` are constant, and each element has at most six neighbors, these arithmetic operations are offloaded to the host CPU, as discussed in Section 7.1.5. The results are stored inside the look-up tables, implemented as discussed in Section 7.1.4.

If look-up tables are not used, the results from offloaded arithmetic operations can be stored directly in each memory block. However, while the memory block performs the computation, it cannot read data from off-chip memory. This makes it a less desirable solution, primarily when pipeling is implemented for further optimization, which will be discussed in Section 7.3.3.

7.2.2.2 Execution Timeline for Elastic Wave Simulations

The volume and flux kernels for elastic wave simulation have a similar execution timeline as the acoustic wave simulation. However, a single memory block is not sufficient to hold single element in the case of elastic wave simulation, as discussed earlier in Section 7.2.1. The `variables` are split into four memory blocks; thus, data exchange between adjacent memory blocks is inevitable. The execution timeline will be discussed when explaining an optimization technique called expansion in Section 7.3.2.

7.3 Optimizations for Wave Simulation

While the basic implementation of wave simulation in PIM can already be used to run acoustic wave simulations, it is insufficient for running elastic wave simulations. This section introduces several optimization techniques I performed to improve the wave simulation implementation in PIM. The first optimization aims to make the PIM implementation scalable through batching, allowing it to handle larger problem sizes that exceed the capacity of the PIM chip, as discussed in Section 7.3.1. The second optimization deals with expanding an element into multiple memory blocks. This technique is useful to increase parallelism for small problem sizes and is required for elastic wave simulation, as discussed in Section 7.3.2. The third optimization intends to increase throughput and hide communication overhead between memory blocks by pipelining, as discussed in Section 7.3.3.

7.3.1 Handling Large Problem Sizes with Batching

The basic implementation discussed in Section 7.2 assumes that the PIM can have as many memory blocks as possible, realizing unlimited total capacity. However, the size of each memory block is limited. This is an unrealistic assumption; thus, in real-world scenarios, PIM can only handle problem sizes that can fit inside the PIM chip. For example, with refinement level 5 and 32,768 elements, the minimum capacity of the PIM chip to run acoustic wave simulation is 4 GB. With a technique called batching, PIM with a smaller capacity than 4 GB can still handle the same problem size by processing part of the problems at one time, called batch, while keeping the rest stored in off-chip memory (Section 7.1.5).

7.3.1.1 Batching for Volume and Integration Kernels

Since the operations are local to each element, performing batched computation is simple for volume and integration kernels. Thus, there is no need for data exchange between the memory blocks. However, as discussed earlier, this is true only

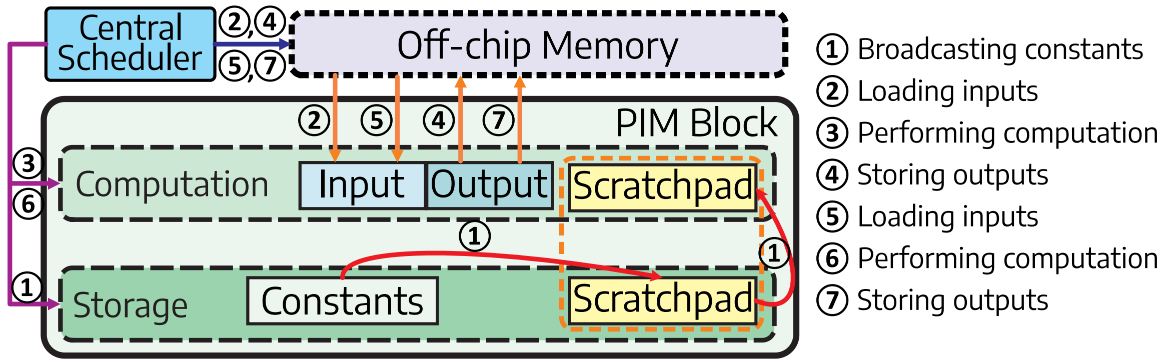


Figure 7.7: The batching mechanism for volume and integration kernels, showing an example involving mesh divided into two sub-meshes. Step ① broadcasts constants used by all sub-meshes, while subsequent steps are to load one sub-mesh, perform computation, and store the result.

for acoustic wave simulation, where one memory block can hold one element. In brief, the mesh is divided into several sub-meshes, and PIM loads one sub-meshes from off-chip memory using memory data layout explained in Section 7.2.1, performs computation using the execution timeline explained in Figure 7.6, and stores the results back to off-chip memory.

Figure 7.7 shows how batching is performed for mesh divided into several sub-meshes. The steps ① to ④ process the first sub-meshes using the execution flows described in Figure 7.6. For subsequent sub-meshes, step ① (i.e., broadcasting constant) can be skipped since the constants (e.g., derivative of shape functions, GLL point, GLL weight), as the name suggests, do not change throughout the run and equal for all elements. Only data that is unique to each sub-mesh needs to be loaded. The overhead of batching is two additional transactions per subsequent sub-mesh: storing the results of the preceding sub-mesh and loading data of the next sub-mesh.

7.3.1.2 Batching for Flux Kernel

Implementing the batching mechanism in the flux kernel is significantly more complicated since the operations are no longer local to each element. Two neighboring

elements must be guaranteed to reside on PIM chip to run the flux kernel. Figure 7.8 shows an example of performing batched flux computation for refinement level 5 with 32,768 elements (i.e., 32×32×32) on a 2 GB PIM chip. In this case, the mesh is divided into two sub-meshes since storing them inside the PIM chip is insufficient.

Consider the sub-meshes are sliced along the y-axis into 16 slices each. Slices 0 to 15 are parts of the first sub-mesh and reside inside the PIM chip, while slices 16 to 31 are parts of the second sub-mesh and reside inside the off-chip memory. Flux computation cannot be performed for slices 15 and 16 since one of them is not inside the PIM chip. This is the edge case that has to be considered when implementing batching in the flux kernel.

In 3D space, there are three axes (i.e., x-axis, y-axis, and y-axis), two normal vectors per axis (i.e., -1 and $+1$), and up to six neighbors for each element, except for elements located on the boundary. Following the slicing along the y-axis, as illustrated in Figure 7.8, the data of neighbor elements along faces located on the x-axis and z-axis[4] are always available since they are loaded to the PIM chip as one slice. Then, the flux computation can be performed quickly, shown as steps ② and ③ for slices 0 to 15 and steps ⑧ and ⑨, following execution flow as described in Figure 7.6.

Unfortunately, the execution flow for computing flux for neighbor elements along faces on the y-axis is tricky. Supposed that, for elements that want to perform flux computation on its face 2 with -1 normal vector (i.e., its neighbor has face 3 with $+1$ normal vector), they can be grouped with their neighbor, called negative group: $(0, 1), (2, 3), ..., (30, 31)$. Likewise, for elements that want to perform flux computation on its face 3 with $+1$ normal vector (i.e., its neighbor has face 2 with -1 normal vector), they can be grouped with their neighbor, called positive group: $(1, 2), (3, 4), ..., (29, 30)$.

Computing flux for the negative groups for the first sub-mesh (i.e., slices 0

[4]i.e., face 0 (-1) and face 1 ($+1$) are along the x-axis. In contrast, face 4 (-1) and face 5 ($+1$) are along the z-axis. See Figure 4.4 for face numbering.

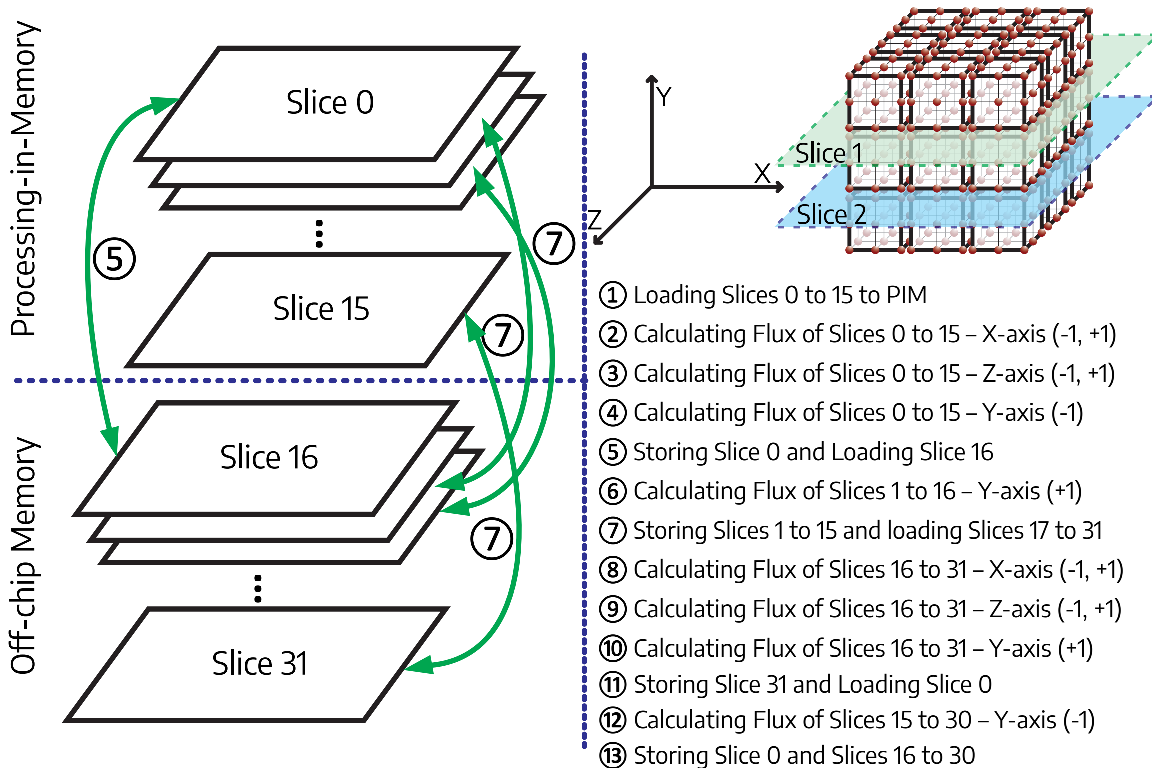


Figure 7.8: The batching mechanism for flux kernel, showing an example involving mesh divided into two sub-meshes. The sub-meshes are divided into several slices along the y-axis. Step ① loads the first sub-mesh to PIM. Step ② and ② perform the flux computation for faces along x-axis and z-axis. The special case must be handled for the interface between the sub-meshes: the face between the elements on the last slice of the first sub-mesh and the first slice of the subsequent sub-mesh (i.e., steps ⑤ and steps ⑪).

to 15) can be performed directly since there is no interaction between slices 15 and 16 (i.e., step ④). However, there is interaction between slices 15 and 16 when computing flux for the positive groups. To alleviate the issue, slice 16 is also loaded to the PIM chip, allowing the flux computation to continue (i.e., step ⑤). Slice 0 must be stored back to the off-chip memory to make some room for storing slice 16. Then, flux can be computed for positive groups (i.e., step ⑥).

To proceed with the second sub-mesh, the slices 1 to 15 are stored back to the off-chip memory, and the slices 17 to 31 are loaded into the PIM chip (i.e., step ⑦).

Note that slice 16 is already inside the PIM chip. Then, the flux computation for the positive groups for the second sub-mesh can be performed directly since there is no interaction between slices 16 and 15 (i.e., step ⑩). To compute the flux for negative groups, slice 31 is stored back in the off-chip memory to make room for loading slice 0 to the PIM chip (i.e., step ⑪), allowing the computation to be performed (i.e., step ⑫). Finally, the results are stored back in the off-chip memory (i.e., step ⑬). In this way, flux computation of large problem sizes can be performed with a PIM chip with smaller capacity.

7.3.2 Increasing Parallelism with Expansion

Although batching allows the PIM chip to handle problem sizes larger than its capacity, it still relies on the assumption that one element can fit into one memory block. This means it will not work straightforwardly for elastic wave simulation or simulation runs that use a double-precision floating point format, where the element cannot fit inside one memory block. In addition, for problem sizes smaller than the capacity of the PIM chip, some of the memory blocks are not utilized, lowering the parallelism that the PIM chip can achieve. This section discusses an optimization technique called expansion, with which an element can be split and stored into multiple memory blocks. Expansion is useful for getting more parallelism out of the PIM chip for small problem sizes and is especially required for running elastic wave simulations.

7.3.2.1 Expansion in Acoustic Wave Simulation

As discussed earlier in Section 7.2, an element can be fit into one memory block for the acoustic wave simulation, and the four contributions are computed serially. For small problem sizes, many available memory blocks are not utilized to perform the computation. For example, refinement level 4 mesh deployed on a PIM chip with 2 GB capacity only occupies 25% of available memory blocks, leading to low utilization

of PIM resources and limiting attainable performance due to less parallelism. The idle memory blocks also consume static power due to leakage current.

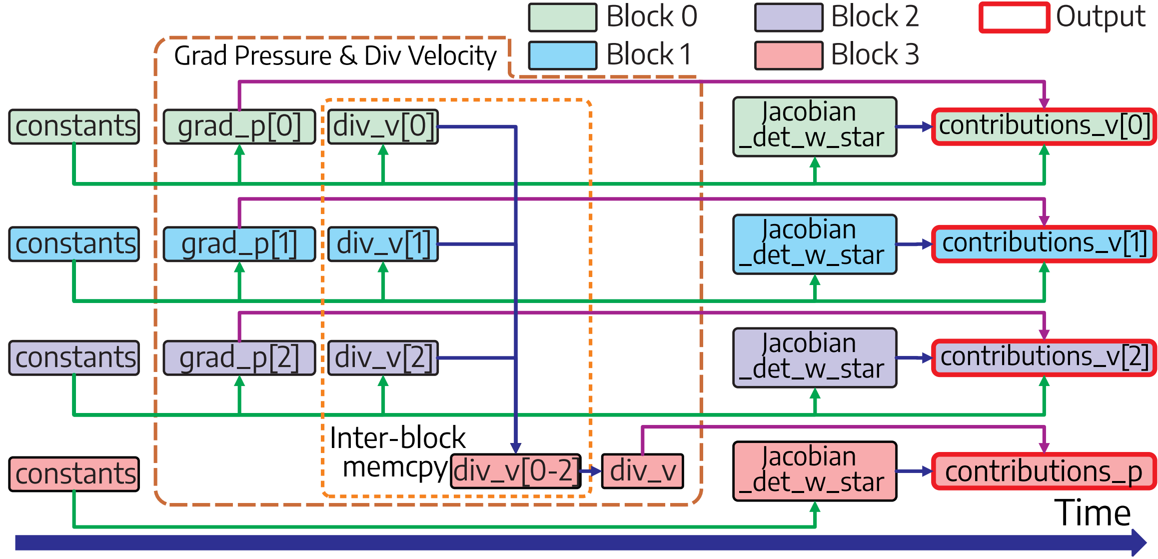


Figure 7.9: The execution timeline for volume kernel using an optimization technique called expansion, leveraging four memory blocks to compute each contribution in parallel. While some data duplications and exchanges between memory blocks are needed, expansion yields better parallelism, improving overall performance.

Since the computation of each contribution (i.e., p, $\mathbf{v}_x$, $\mathbf{v}_y$, and $\mathbf{v}_z$) can be performed individually in parallel, it is possible to perform the calculation simultaneously by using four memory blocks instead of one. However, some data duplications must be done by copying commonly-used data to four memory blocks. These data include `variables`, `materials`, and some constants. In addition, there may be additional overheads of data exchange between these four memory blocks, particularly for sharing intermediate results used by four memory blocks to compute the `contributions`. Below are the implementations of expansion on integration, volume, and flux kernels.

- *Integration kernel.* Implementing expansion to the integration kernel is straightforward since no intermediate result needs to be shared between four memory blocks. Updating single variable of a node only needs data on that particular variable and particular node from `contributions`, `mass_inverse`, and

auxiliary. The updated variable is stored inside the respective memory block (i.e., the variables are split into four memory blocks).

- *Volume kernel.* Although the volume kernel only involves local operations within an element, splitting the contributions into four memory blocks generates data exchanges between them. Thus, it is more complicated to implement expansion in volume kernel compared to integration kernel. There are some modifications to the basic execution timeline of the volume kernel (i.e., the top part of Figure 7.6), as illustrated in Figure 7.9. First, the jacobian_det_w_star must be calculated four times, one in each memory block, and constants must be duplicated into four memory blocks. While the intermediate results grad_p remain in their respective memory block, the intermediate result div_v must be transferred and gathered into the memory block responsible for computing the contribution of p. Although expansion yields better parallelism and improves performance, it consumes more dynamic power than single-memory block implementation.

- *Flux kernel.* Figure 7.10 illustrate the implementation of expansion on the flux kernel across four memory blocks. One memory block is dedicated to buffering the required neighboring data (i.e., neighbor's variables and materials). Buffering neighbor's data on an adjacent memory block reduces the latency data transfer instead of getting them from a distant memory block. This is especially important when using the H-Tree interconnect topology. The other three memory blocks are used for computing the flux on the x-axis, y-axis, and z-axis in parallel. While transferring the neighbor element's data to the buffer takes time, it can be overlapped with computations that do not require it, such as computing jacobian_det_w_star.

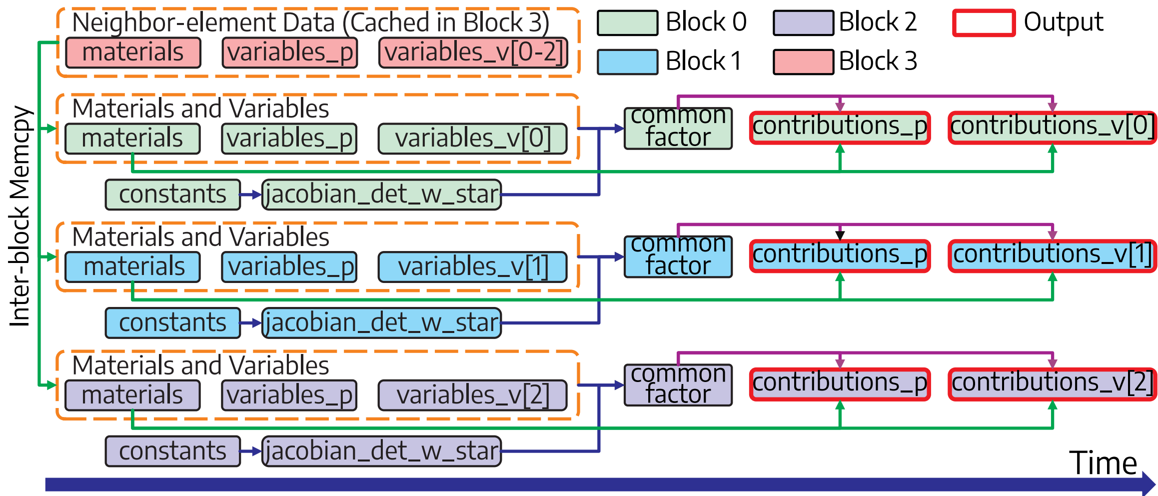


Figure 7.10: The execution timeline for flux kernel using an optimization technique called expansion, leveraging four memory blocks. One memory block buffers neighboring elements' data to reduce communication latency, leveraging the property of the H-Tree interconnect topology. In contrast, three memory blocks are used to compute the flux contribution in each axis in parallel.

7.3.2.2 Expansion in Elastic Wave Simulation

Instead of optionally using expansion to improve parallelism, yielding higher performance, elastic wave simulation requires using expansion to split the element into four memory blocks since one memory block cannot fit single element. The nine variables of elastic wave simulations can be divided into three or, more aggressively, nine memory blocks. The execution timeline for flux and integration kernels for elastic wave simulations follows the same manner as the acoustic wave simulations. For volume kernel, it follows the execution timeline given in Figure 7.9 with the same data transfer pattern between memory blocks albeit with higher data volume.

7.3.3 Improving Throughput with Pipelining

For acoustic wave simulation, further execution flow optimization can be done using pipelining. On the other hand, it is more challenging to implement pipelining for elastic wave simulation due to the amount of data involved and the need to split

single element into multiple memory blocks.

Following the approach to dedicate one memory block to buffering neighboring elements' data, a pipeline for flux kernel can be implemented in six stages, corresponding to three axes and two normal vectors per axis. When computing flux contribution, there is no data dependency between different axes and normal vectors. On the other hand, there is data transfer and data broadcast inside the memory block for volume and flux kernels, as shown previously in Figure 7.5. This intra-memory-block data transfer makes pipelining impossible for volume and flux kernels since it is a hardware hazard; memory blocks cannot perform computation and data movement simultaneously since they both involve applying different voltages on bit lines and word lines.

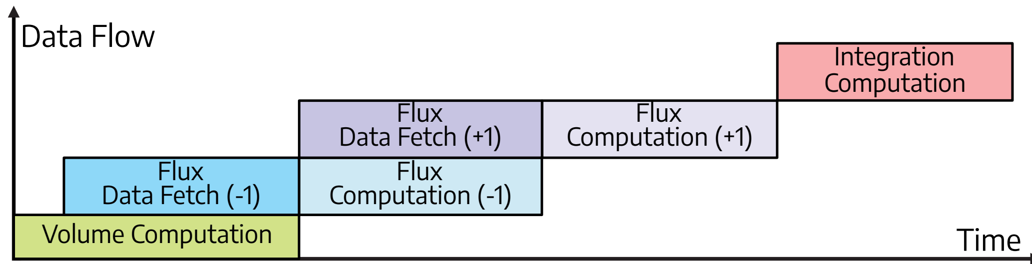


Figure 7.11: The execution flow of wave simulation combining pipelining and expansion optimization techniques. The volume kernel can be overlapped with data fetching for flux kernels. Using expansion, computing flux contributions on each axis can be performed in parallel, resulting in two stages for the flux kernel. Finally, the integration kernel can only be performed after calculating all volume and flux contributions.

Further optimization for the pipelining strategy is to overlap the execution of two stages if there is no data dependency between them. For example, fetching neighboring elements' data in the flux kernel can be overlapped with volume kernels by executing them in parallel. In addition, the pipelining technique can also be used with the expansion technique in the flux kernel. The three axes can be processed simultaneously for the flux kernel, allowing the construction of two pipeline stages for flux kernels based on two normal vectors instead of the six described earlier. Finally,

the integration kernel can only be executed once all volume and flux contributions have been computed. These two optimizations for the pipelining technique result in execution flow given in Figure 7.11.

7.4 Performance Analysis

This section discusses the performance evaluation of the proposed PIM system for wave simulations. First, the configurations for the PIM system are briefly discussed, with some already covered in Section 3.3. Secondly, PIM's performance and energy efficiency are compared against the GPU implementation described in Chapter 5. These evaluations include the optimization techniques, such as batching and expansion. Finally, the efficacy of the features implemented inside the PIM, which include the pipelining, the memory block interconnect topology, and the arithmetic accuracy, are evaluated.

7.4.1 Configurations for Evaluation

Before discussing the performance analysis of the PIM systems, the configurations for performance evaluation must be established. This includes defining the workloads for benchmarking, establishing the baseline, and determining the configuration of the PIM systems. The experiments are run in single-precision floating-point format (Section 7.1.6).

7.4.1.1 Workloads for Benchmarking

The benchmark used for the performance evaluation of the PIM system is the acoustic and elastic wave simulations described in Chapter 4, with detailed explanation in Appendix B. There are two types of problems: acoustic (Section 2.2.1) and elastic (Section 2.2.2) wave simulation. The acoustic wave simulation only has a Riemann flux solver, while the elastic wave simulation has either Central or Riemann flux solvers (Section 4.4.3 and Appendix B.4.3). Each has two problem sizes: refinement

level 4 with 4096 elements and refinement level 5 with 32768 elements. The total combinations are six workloads, as shown in Table 7.2. The benchmark naming is as follows: The number added as a suffix to the benchmark's name denotes the refinement level; acoustic benchmarks use a Riemann flux solver; and elastic benchmarks use a Central or Riemann flux solver.

Benchmark	**Refinement Level**	**Number of Elements**	**Number of Instructions**	**Number of FP Ops.**
Acoustic_4	4	4,096	2,140,930,048	391,380,992
Elastic-Central_4	4	4,096	3,465,543,680	990,117,888
Elastic-Riemann_4	4	4,096	9,870,131,200	1,472,200,704
Acoustic_5	5	32,768	17,127,440,384	3,131,047,936
Elastic-Central_5	5	32,768	27,724,349,440	7,920,943,104
Elastic-Riemann_5	5	32,768	78,960,159,424	11,777,661,440

Table 7.2: The characteristics of the workloads used for evaluating PIM performance consist of acoustic and elastic wave simulations with different flux solvers and refinement levels. The instruction count and the Floating-point operations are obtained from `GPU_fl` kernels launched once.

The workloads have a total number of instructions in the range of 2 billion to 79 billion, with 400 million to 12 billion single-precision floating-point operations. These numbers are obtained using `nvprof` described in Section 3.4.1 for `GPU_fl` kernels, as explained in Section 5.5. The kernels are launched once on the NVIDIA Tesla V100 GPU. In real-world scenario, the simulation runs for thousands of time steps, where the kernel is launched five times per time step (Section 4.3.1).

7.4.1.2 Establishing Baseline

The next step is to establish the baseline for comparing PIM's performance and energy efficiency. The basic implementation of GPU kernels (`GPU_base`) described in Section 5.3 is used as the baseline GPU code while the NVIDIA GeForce GTX 1080 Ti GPU is used as the baseline GPU hardware since it is the slowest out of the three available GPUs in TACC Maverick2 cluster (Table 3.1). Therefore, `GPU_base` running on NVIDIA GeForce GTX 1080 Ti GPU is the baseline for performance evaluation comparing PIM against GPU implementation of wave simulations. To recap,

the `GPU_base` launches volume and flux kernels individually with basic optimizations such as storing repeatedly used values in GPU's constant memory (Section 2.6.3), rearranging codes to minimize thread-divergence, and extracting element-level and node-level parallelism.

In addition to the `GPU_base`, the performance evaluation also considers more optimized GPU implementation by applying techniques described in Sections 5.1.2.2 and 5.4.1. This optimized GPU kernel is called `GPU_fl`, as explained in Section 5.5. This optimized GPU kernel merges volume and flux into one kernel to minimize the data movement between off-chip and on-chip memory. It also incorporates more efficient look-up tables for determining neighboring elements and better data locality for each thread by dedicating each to handle one node throughout kernel execution. To make an apple-to-apple comparison, all GPU kernels are configured to run in single-precision floating-point arithmetic.

7.4.1.3 PIM Configuration Flavors

Based on the basic implementation discussed in Section 7.2, the optimization techniques described in Section 7.3, the PIM platform configurations given in Table 3.5, and the workloads configurations given in Table 7.2, there are several PIM configuration flavors, as summarized in Table 7.3. All configurations run single-precision (32-bit) floating-point arithmetic and offload complex arithmetic operations to the host CPU (Section 7.1.5). Structure-wise, they implement H-tree or Bus routing switches as interconnect topology for memory blocks and interconnect topology for memory tiles (Section 7.1.3). They also have one central controller to oversee the operation of the PIM chip and additional decoder logic inside each memory block to implement custom ISA instructions (i.e., for implementing LUT on PIM, see Section 7.1.4).

In Table 7.3, N denotes basic implementation of PIM, as described in Section 7.2. The E_a denotes expansion aimed to increase the parallelism, as described in

Configuration	PIM-512	PIM-2G	PIM-8G	PIM-16G
Acoustic_4	N	E_a	E_a	E_a
Elastic-Central_4	E_e&B	E_e	E_a&E_e	E_a&E_e
Elastic-Riemann_4	E_e&B	E_e	E_a&E_e	E_a&E_e
Acoustic_5	B	B	N	E_a
Elastic-Central_5	E_e&B	E_e&B	E_e&B	E_e
Elastic-Riemann_5	E_e&B	E_e&B	E_e&B	E_e

Table 7.3: The PIM configuration flavors for performance evaluation on different benchmarks and PIM chip capacities. The N, E, and B denote basic implementation, expansion, and batching techniques, respectively. The subscript e denotes the expansion technique for insufficient memory block, which is only applicable for elastic wave simulations, while subscript a denotes the expansion technique for increasing parallelism, which is applicable to both acoustic and elastic wave simulations.

Section 7.3.2. E_a can be used for both acoustic and elastic wave simulations. The E_e denotes expansion required for elastic wave simulations due to insufficient memory block to hold a single element (i.e., only exists in elastic wave simulations). Finally, B denotes the batching technique applicable to acoustic and elastic wave simulations whenever the PIM chip capacity is insufficient to hold larger problem sizes.

For acoustic wave simulation with refinement level 4, basic implementation (N) is sufficient to run on PIM-512. To take advantage of larger capacity chips (i.e., PIM-2G, PIM-8G, and PIM-16G), expansion techniques for increasing parallelism (E_a) can be applied to boost performance. For acoustic wave simulation with refinement level 5, batching technique (B) is required for PIM-512 and PIM-2G chips due to insufficient capacity to hold all elements. Basic implementation (N) is the only available option for PIM-8G, while expansion techniques for increasing parallelism (E_a) can be applied to boost performance on PIM-16G.

All elastic wave simulations for refinement levels 4 and 5 require an expansion technique applied due to insufficient memory block to hold single element (E_e). The batching technique (B) is necessary for PIM-512 for both refinement levels due to the insufficient capacity of the PIM chip. Refinement level 4 can run on PIM-2G without batching, while refinement level 5 needs batching (B). Increasing the PIM chip capacity to PIM-8G allows applying expansion for increasing parallelism (E_a)

to refinement level 4, while refinement level 5 still requires batching (B). Finally, PIM-16G can hold refinement level 5 without batching, while refinement level 4 can enjoy increased parallelism thanks to expansion technique (E_a).

7.4.2 Performance Comparison Against GPU

This section discusses the performance of the PIM system compared to the baseline GPU (i.e., `GPU_base` on NVIDIA GeForce GTX 1080 Ti), as well as compared to more optimized GPU implementation (`GPU_fl`) running on three different GPUs available on TACC Maverick2 cluster (Table 3.1). Although Section 3.3.5 describes the process node scaling for PIM for a fair comparison with GPUs, both unscaled (original) and scaled performance numbers are compared. In addition, both batching and expansion techniques are also considered, following Table 7.3. Only GPU implementation described in Chapter 5 is considered; comparison with more optimized GPU implementation described in Chapter 6 can be easily done by looking at relative performance improvements across different optimization, shown in Figure 6.16, and scale the relative performance against PIM.

The performance comparison of PIM against the GPU is given in Figure 7.12. Thanks to the reduction in data movement, PIM can achieve up to 414.37×speed-up over the GPU implementation. On average, PIM-512, PIM-2G, PIM-8G, and PIM-16G achieve 10.28×, 35.80×, 72.21×, and 172.76× speed-up over `GPU_base` running on NVIDIA GeForce GTX 1080 Ti GPU across six benchmarks, respectively. Compared to the most efficient GPU implementation, `GPU_fl` running on NVIDIA Tesla V100 GPU, PIM-512, PIM-2G, PIM-8G, and PIM-16G achieve 2.30×, 7.89×, 15.97×, and 37.39× speed-up, respectively.

The achieved speed-up of PIM over GPU on Elastic-Riemann is below average since this benchmark has the highest arithmetic intensity. Thus, the performance improvements due to data movement reduction are insignificant compared to the time spent performing the computation. In addition, the PIM-512 performed worse

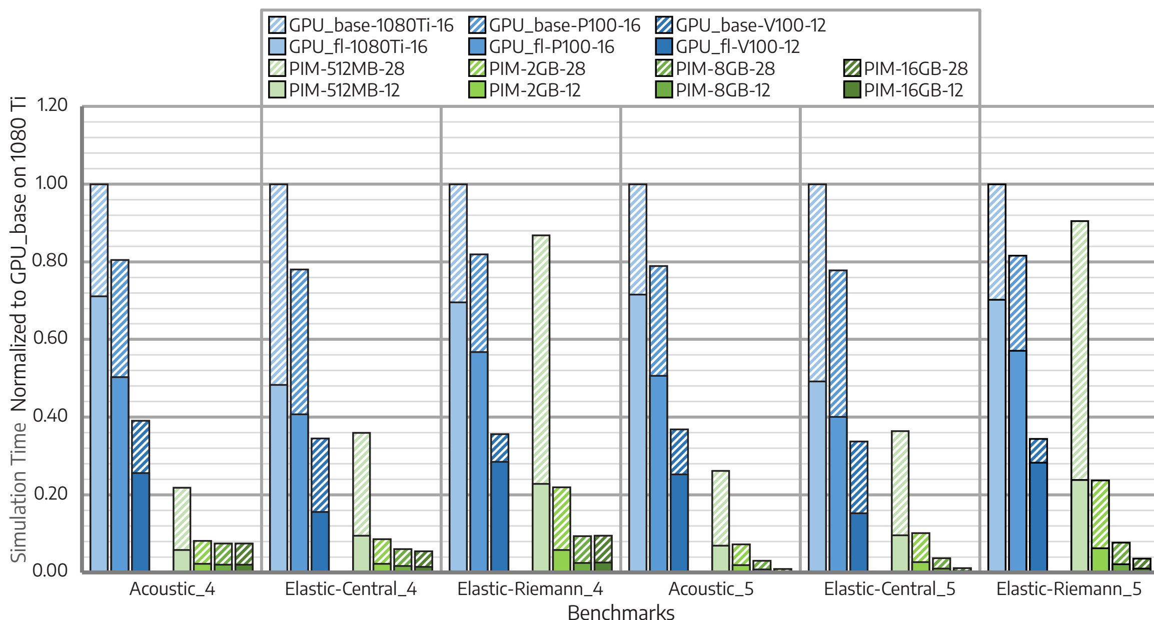


Figure 7.12: The performance comparison of PIM against GPU across six benchmarks. Both unscaled (striped green) and scaled (solid green) numbers for PIM are reported. The GPU implementation uses either `GPU_base` (striped blue) or `GPU_fl` (solid blue) kernels, running on three different GPUs. Depending on problem type, problem size, and PIM chip capacity, the PIM implementation uses configuration flavors described in Table 7.3. The performance number is normalized against `GPU_base` running on NVIDIA GeForce GTX 1080 Ti.

due to the additional data movement between the PIM chip and the off-chip memory due to the batching technique, which divides the problem with refinement level 5 into 32 batches.

7.4.3 Energy Consumption Comparison Against GPU

This section discusses the energy consumption of the PIM system compared to the baseline GPU (i.e., `GPU_base` on NVIDIA GeForce GTX 1080 Ti), as well as compared to more optimized GPU implementation (`GPU_fl`) running on three different GPUs available on TACC Maverick2 cluster (Table 3.1). Although Section 3.3.5 describes the process node scaling for PIM for a fair comparison with GPUs, both unscaled (original) and scaled energy consumption numbers are compared. In addition,

both batching and expansion techniques are also considered, following Table 7.3.

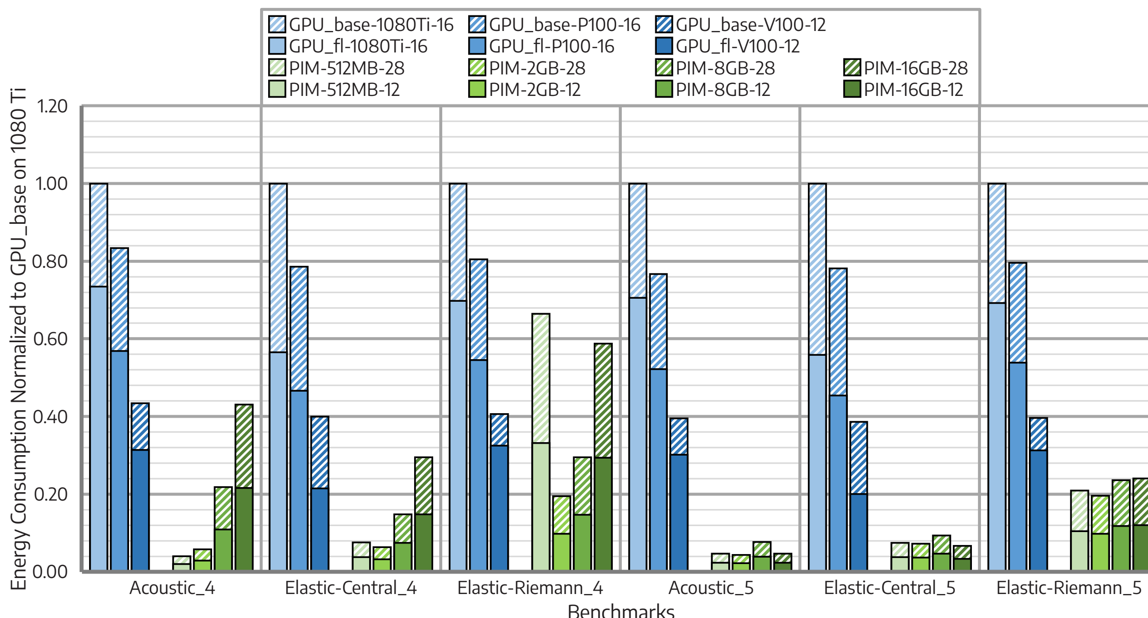


Figure 7.13: The energy consumption comparison of PIM against GPU across six benchmarks. Both unscaled (striped green) and scaled (solid green) numbers for PIM are reported. The GPU implementation uses either `GPU_base` (striped blue) or `GPU_fl` (solid blue) kernels, running on three different GPUs. Depending on problem type, problem size, and PIM chip capacity, the PIM implementation uses configuration flavors described in Table 7.3. The energy consumption number is normalized against `GPU_base` running on NVIDIA GeForce GTX 1080 Ti.

The energy consumption comparison of PIM against the GPU is given in Figure 7.13. If the capacity of the PIM chip is sufficient to store the whole problem without the need for batching, PIM achieved up to 50.56× energy savings compared to the GPU implementation. On average, PIM-512, PIM-2G, PIM-8G, and PIM-16G achieve 26.62×, 26.82×, 14.28×, and 16.01× energy savings compared to `GPU_base` running on NVIDIA GeForce GTX 1080 Ti across six benchmarks.

The larger capacity of PIM chips benefits large problem sizes due to the abundant parallelism and zero data movement overhead between PIM chips and off-chip HBM2 memory since batching is not required. However, small problem sizes cannot take advantage of the larger capacity of the PIM chip, leading to resource under-

utilization. The higher leakage power associated with more numbers of memory blocks further reduces energy savings achieved by the larger PIM chip compared to the smaller one. Hence, there is a trade-off between performance and energy efficiency.

7.4.4 Pipelining Analysis and Evaluation

This section discusses the execution timeline of the pipelining technique explained in Section 7.3.3. The execution of the volume kernel can be overlapped by the computation of offloaded arithmetic functions on the host CPU since the volume kernel does not need it. While the volume kernel is still executing, data fetching for the first part of the flux kernel (i.e., positive normal vector) can run. The data fetching transfers the neighboring elements' data to the buffer, as shown in Figure 7.10.

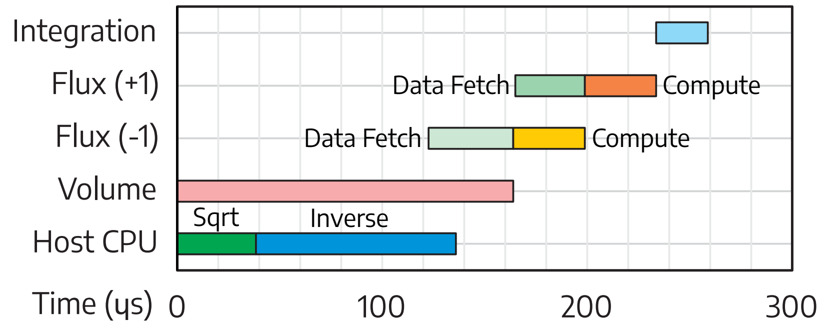


Figure 7.14: The execution time breakdown of the PIM system as a result of optimization through pipelining. The execution of the volume kernel can be overlapped with the host CPU, computing complicated arithmetic functions; the data fetching for the first part of the flux kernel can be overlapped with part of the volume kernel after the host CPU finishes computing the offloaded arithmetic functions. Finally, the data fetching for the second part of the flux kernel can be overlapped by the computation of the first part of the flux kernel. Integration kernel cannot be overlapped.

The computation for the first part of the flux kernel can run once the offloaded arithmetic functions return results and the data fetch finishes. This can also be overlapped by data fetching for the second part of the flux kernel (i.e., negative normal vector). The second part of the flux kernel can run once the data fetch finishes, and the first part concludes its execution. The integration kernel cannot be overlapped

since the contributions must be ready before launching this kernel. Based on this diagram, without pipelining and overlapping the execution of multiple kernels, the PIM can only achieve 0.77× performance shown in Figure 7.12.

7.4.5 Memory Block Interconnect Evaluation

This section compares H-Tree and Bus interconnect to connect memory blocks, as discussed in Section 7.1.3. If there are few data exchanges between memory blocks, such as for volume and integration kernels, the performance of both interconnects is similar. However, the Bus interconnect consumes less power than the H-tree. On the other hand, if there is an intensive data exchange between memory blocks, the performance difference between these two interconnects becomes significant. Figure 7.15 shows the performance comparison of H-Tree and Bus interconnects when executing flux kernel, which involves many data transfers between memory blocks, for refinement level 4.

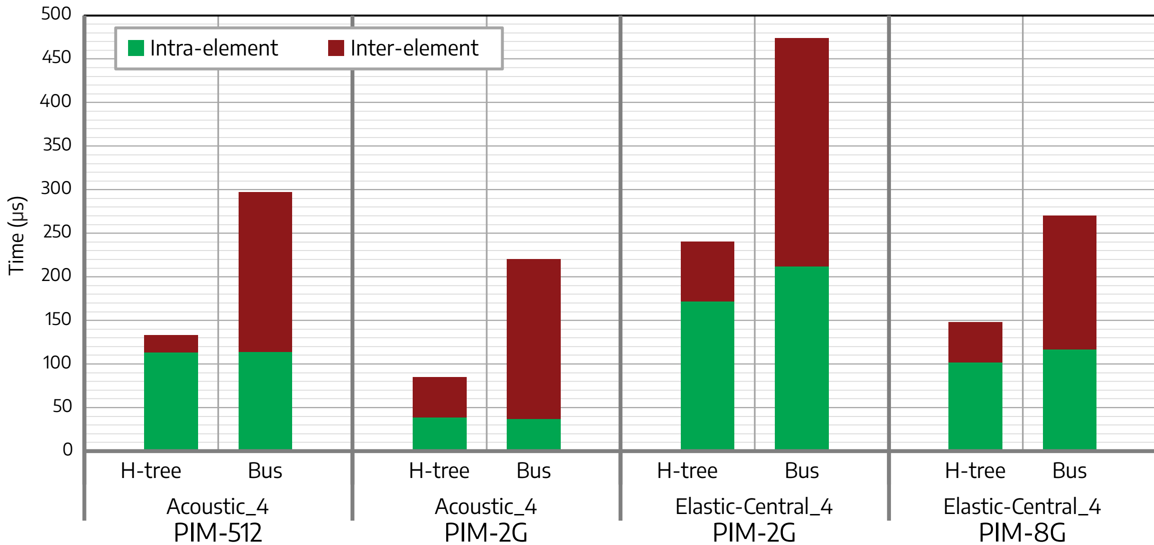


Figure 7.15: The comparison of H-Tree and Bus interconnect between memory blocks when executing flux kernel, showing the breakdown of intra-element (i.e., due to expansion) and inter-element communication (i.e., due to flux computation) for refinement level 4.

Without expansion (i.e., Acoustic_4 on PIM-512 and Elastic-Central_4 on PIM-2G), the inter-element data transmission contributes 21.62% and 58.41% to the overall execution time for the H-Tree and Bus interconnects, respectively. If expansion is applied (i.e., Acoustic_4 on PIM-2G and Elastic-Central_4 on PIM-8G), the inter-element data transmission contributions are higher: 42.77% and 69.96% for H-Tree and Bus interconnects, respectively. To be able to implement the pipelining mechanism shown in Figure 7.14, the time spent for inter-element communication must be shorter than intra-element communication, and thus, H-tree interconnection is the one used in the design.

Chapter 8: Conclusion

This chapter recaps the key findings from this dissertation already discussed in a detailed fashion in Chapters 5 to 7. First, each major finding is summarized in Section 8.1, emphasizing their importance in the relevant fields. Secondly, the key takeaways are summarized in Section 8.2. Finally, Section 8.3 identifies potential ideas for future research based on the gaps or limitations of the research done in this dissertation.

8.1 Summary

This dissertation explores and characterizes the key bottlenecks in a class of wave simulations that use the dG finite element method (Section 2.3.1) with a GLL integration scheme (Section 2.3.2) on hexahedral elements with straight faces, decreasing the overall required BLAS operations and simplifying them into Level-1 BLAS routines. Although the computational cost of these wave simulations is significantly lower than those performed on general meshes, they remain too expensive for many high-fidelity, industry-relevant applications. Therefore, accelerating time-to-solutions and improving energy-to-solutions is crucial since these simulations are often run to find millions of wave solutions. Furthermore, many existing implementations use general-purpose CPUs in large computing clusters to handle large problem sizes, such as the one discussed in Chapter 4. However, CPUs may not be the most efficient hardware for running wave simulations since they have considerable predictability in the execution flow, regularity in memory accesses, and, most importantly, the abundant parallelism that can be extracted.

The first contribution of this dissertation is accelerating the dG-based wave simulations using Graphics Processing Units, discussed in Chapter 5 and published as a paper by Hanindhito et al. (2022). However, converting CPU codes to GPU codes

and achieving satisfactory performance is not trivial; hardware-informed optimization on GPU must be performed while preserving numerical accuracy and stability. Furthermore, modifications must be performed to the fundamental parts of the existing CPU codes to make them more suitable for acceleration using GPUs. The modification includes storing and handling the mesh data and structure on GPUs since the mesh handling library used by the existing CPU codes does not support execution on GPU. While the simulation and data flow largely follow the CPU codes, some modifications are done to allow for as much parallelism extraction as possible.

Lowering computational costs using the abovementioned meshes and integration schemes proportionally amplifies the impact of data movement even though dG has lower communication costs than FDM and SEM, making attaining peak performance of the GPU difficult. This is evident from the characterization of the basic implementation of the GPU kernels. Even though they achieve significant performance improvements over the CPUs, their performance is limited due to the excessive data movement between on-chip and off-chip memory, making them memory-bound workloads. As a part of the first contributions, multiple optimization strategies are implemented, resulting in significant performance gains across different wave-simulator configurations. By applying kernel fusion and LUT-based neighbor search optimizations, the optimized wave simulations achieved up to a 2.6× speed-up compared to the basic GPU implementation. Optimizing shared memory usage and using SM-occupancy-aware register allocation further boosts the performance by 1.49×. However, with these optimizations, the wave simulations are still memory-bound workloads.

Another part of the first contribution is making the GPU implementation scalable, allowing it to support large problem sizes by utilizing multiple GPUs in multiple HPC compute nodes. Instead of writing a mesh handling library from scratch, custom functions are implemented to perform the data transfer directly from the GPUs. However, another problem arises when inter-device communication interfaces are used; the inter-node communication links are often the weakest in computing clusters, potentially limiting the overall wave simulation performance. Therefore, optimization

strategies are developed to minimize intra- and inter-node communication overhead in multi-GPU, multi-node systems. These strategies include: a) reducing the size of the exchanged data, which resulted in an average reduction of communication time by 70.27%, and b) utilizing an MPI implementation with support for GPUDirect RDMA, GPUDirect P2P, and asynchronous progression, further cutting overhead by 82.03%. Ultimately, these approaches allow the wave simulations to achieve weak scaling across 128 GPUs.

The second contribution of this dissertation is developing algorithms to reduce the communication overhead in GPU-accelerated dG-based wave simulations, discussed in Chapter 6. Two parts of the communication problem are addressed in this contribution: the intra-device and the inter-device communications. It can be viewed as an extension to the Chapter 4, with more advanced algorithms to address the shortcomings of previous GPU implementations.

The first part of this contribution is to develop algorithms to reduce intra-device communication. The Node-tiling strategy delivers an average performance improvement of 20%, with up to 37% better performance compared to the kernel without node-tiling. Incorporating shared memory and improved register allocation further boosts the performance of the node-tiling kernel by an additional 15%. Furthermore, using a unified kernel enhances performance by an average of 22%. Using shared memory to store contributions and optimize register allocation brings another 11% average performance improvement. Storing both contributions and variables in shared memory adds a further 7% improvement on average.

The second part of this contribution is to develop algorithms to reduce inter-device communication. The Face-Node-Only ghost exchange results in lower inter-device communication overhead with an average of 82% and 41% reduction compared to the previous implementations. On the other hand, the reduced-precision ghost exchange can significantly cut the size of ghost elements by 50% when moving from double precision to single precision, and by 75% when going to half precision. This

results in an average reduction in ghost exchange overhead of 74.06% and 49.17%, respectively. Although using lower-precision elements for ghost exchange leads to higher L_2 errors, all wave simulations conducted during the experiments were completed without any numerical instability issues. Lastly, while still in the early stages of exploration, partial ghost exchange shows promise for reducing the amount of data transmitted during ghost exchange, albeit with additional computation. However, it must be validated for stability when run on CPUs before GPU-accelerated implementations are developed.

The third contribution of this dissertation is accelerating the dG-based wave simulations using memristor-based Processing-in-Memory, an emerging computing technology, discussed in Chapter 7 and published as a paper by Hanindhito et al. (2021). Processing-in-Memory (PIM) is an evolving computing technology that alleviates the bottleneck between compute units and off-chip memory in von Neumann architectures, enhancing performance and lowering energy consumption. Since PIM is also a massively parallel architecture, it is suitable for extracting the parallelism of the wave simulations while addressing the concerns with the excessive data movement in GPU implementations. However, converting the CPU codes to run on PIM is even more difficult; no compiler or software stack can conveniently convert the applications to utilize PIM. Thus, all the basic implementation and optimizations done in PIM must be performed manually.

The first part of the third contribution is to develop PIM architecture tailored for wave simulations, which includes the interconnect topology and the look-up table implementations. Then, the data must be laid out manually to yield efficient execution while utilizing as much parallelism provided by PIM as possible. The developed data mapping and layout techniques apply to other SRAM- and DRAM-based processing-in-memory technologies. The second part is developing methods to optimize the wave simulations on PIM, extending its scalability, improving performance, and reducing the data movement overhead between memory blocks. Finally, in the third part, PIM's performance and energy efficiency are compared against the GPU

implementation. Together, these optimization efforts led to performance improvements and reduced data movement, lowering energy consumption. Experimental results demonstrate that PIM significantly surpasses GPU implementation described in Chapter 5, achieving an average speed-up of 41.98× and energy savings of 12.66×.

Finally, it is worth mentioning that effective strategies for efficiently computing acoustic and elastic wave equations in GPUs and PIMs can also be utilized for electromagnetic waves due to their structural similarities. All methods, strategies, and techniques presented in this dissertation are relevant for a wider range of applications, underscoring the significance of this dissertation. For instance, kernel fusion is highly applicable to many HPC applications. However, each application may have different register requirements, and thus, fine-tuning the kernel is important, which also applies to the node-tiling. In addition, LUT can also be widely used in many scientific applications, especially where repetitive and complex calculations are beneficial to be replaced by a simple search on an array.

8.2 Key Takeaways

Since the introduction of general-purpose GPUs in 2008, manufacturers have continuously improved their APIs (e.g., CUDA, ROCm) to help users migrate their existing applications running on CPUs to take advantage of them. However, until now, developing the GPU version of the applications is not trivial; it requires significant manual efforts to rewrite the codes, characterize and analyze the bottlenecks, and optimize the applications to take advantage of GPUs optimally. These efforts require a deep understanding of the underlying GPU hardware architecture, which general computational scientists may not possess.

Stanzione (2022) stated that only less than 10% of high-performance scientific codes can actually run on Frontier, a GPU-accelerated supercomputing cluster at Oak Ridge National Laboratory, as described by Atchley et al. (2023); Schneider (2022). While Frontier costs approximately $600 million to build, the Department of

Energy of the United States spent more than $1 billion in exascale software development to harness the computing power of this cluster. The work in this dissertation demonstrates the importance of collaboration between people with algorithm and hardware expertise to develop GPU-accelerated wave simulations. Other work that involves similar collaboration includes work by Wan et al. (2023), who proposes different optimization approaches for extreme-scale Earthquake simulation to obtain high performance on the New-Generation Sunway supercomputer.

While the future development of APIs aims to automate most of these processes by relying on compilers to "convert" CPU codes to "optimized" GPU codes, such collaborations will still be required for many years. For example, the work by Uphoff and Bader (2020) shows such attempts to develop automatic code generation and optimization for domain-specific language aimed at various applications, including DG and SEM.

Moving into processing-in-memory (PIM), application development is significantly more complex than GPU due to the nature of emerging computing technologies. The availability of the software (APIs) and framework is minimal, so all efforts to deploy wave simulations into PIM are done manually. However, using PIM with architectural optimizations tailored for wave simulations, as discussed in Chapter 7, can alleviate, to a certain degree, the excessive intra-device data movement that bothers the GPU, yielding performance and energy efficiency improvements. It will take years, if not decades, for PIM to become widely available and be used in the industry. It is also hard to scale the PIM implementation across multiple PIM chips to support large-scale industry-relevant wave simulations.

8.3 Future Work

Aside from the algorithms, techniques, strategies, and architectures explored in this dissertation, there are abundant opportunities for further accelerating the wave simulations. This section summarizes key exciting ideas to explore as a future

research direction.

First, both kernel fusion (Section 5.4.1) and unified kernel (Section 6.2) are shown to give performance improvements by preserving the GPU execution states and reducing kernel launch overhead. A further idea worth exploring is the persistent kernel, where a kernel runs indefinitely on the GPU. Consider the unified kernel where the wave simulation only has one primary kernel. By moving the two loops of the simulation flow (Figure 6.5) into the GPU, the host CPU only needs to launch one kernel at the beginning of the simulation. This means the single kernel runs the integration and time-stepping loops while performing volume, flux, and integration computations in each iteration. This persistent kernel will keep the GPU execution states throughout the entire simulation. The difficulty comes from the multi-GPU implementation, where the ghost exchange needs to be considered when designing a unified and persistent kernel. Another consideration is simulation checkpointing, allowing the host CPU to dump the simulation data for checkpointing or visualization purposes to disk. Some ideas from previous works by Kim et al. (2022); Gupta et al. (2012); Zhao et al. (2021); Peters et al. (2010) could help implement persistent kernels for wave simulations.

Another interesting idea is the automatic kernel parameters tuning based on the hardware architecture where the kernel is executed. Often, some optimization strategies rely on parameters dependent on the GPU's architecture and the applications' behavior. Instead of burdening the users with finding the optimized kernel parameters through exhaustive tuning, automatic dynamic tuning of the GPU kernel can simplify the deployment of the wave simulations across different GPU architectures. Some ideas from previous works by Guo and Wang (2010); Sato et al. (2010); Lim et al. (2017); Schoonhoven et al. (2023) may be useful for implementing automatic kernel parameter tuning on wave simulation kernels. In addition, exploring techniques to reduce thread divergences on GPUs may be helpful to improve the efficiency of flux computation on the GPU. Previous works have explored various

approaches to reduce the thread divergences, minimizing the impact on overall performance. The ideas presented in previous works by Han and Abdelrahman (2011); Zhang et al. (2010); Han and Abdelrahman (2013); Brunie et al. (2012); Vespa et al. (2015); Fung et al. (2007) may be adopted to reduce the impact of thread divergence during computation of flux contributions.

Furthermore, dynamic parallelism is something that would be exciting to explore. In basic GPU implementation (Section 5.3), volume and flux kernels have different numbers of parallelism: $N_{element} \times N_{nodes_per_element}$ for volume and $N_{element} \times N_{nodes_per_face}$ for flux. When fusing volume and flux kernels by assigning one thread to handle one node, the parallelism in flux is changed to $N_{element} \times N_{nodes_per_element}$. This implies that many threads handling the interior nodes are idle during the flux computation. Dynamic parallelism allows for scheduling different numbers of threads for volume and flux, although both of them are fused into single kernel. First, the kernel launches threads equal to the number of $N_{element}$ to extract element-level parallelism. Then, each thread can launch $N_{nodes_per_element}$ threads for computing volume and $N_{nodes_per_face}$ threads for computing flux. The execution scheme in previous work by Wang and Yalamanchili (2014); Zhang et al. (2015); Wang et al. (2016b); DiMarco and Taufer (2013); Tang et al. (2017) may provide helpful insight on implementing dynamic parallelism for wave simulations on GPUs.

Next, borrowing the idea of the optimized data layout in PIM, discussed in Section 7.2.1, the optimization of how the data is stored inside the GPU memory is interesting to investigate, as it may affect the access pattern of each thread, impacting the performance. For example, the existing implementation stores each variable for each node inside the element data structure as follows. The first variable for nodes 0 to 511 is stored in `variables[0..511]`. Then, the second, third, and fourth variables for nodes 0 to 511 are stored in `variables[512..1023]`, and `variables[1024..1535]`, `variables[1536..2047]`, respectively. This means each thread handling one node has a wide access stride to access all four variables. An alternative data layout is to store all four variables for each node closer to each other. For example, for node 0,

the four variables will be stored in `variables[0..3]`. Likewise, for node 1, the four variables will be stored in `variables[4..7]`. Each thread will have an access stride 1 in this alternative data layout. It would be interesting to investigate whether these simple changes in the data layout will significantly impact performance.

Another idea is to re-run all algorithms, techniques, and strategies on newer GPU architectures. Early exploration on newer GPUs and platforms, such as NVIDIA A100 and DGX-A100, is presented in Sections 5.5.5 and 6.7.1.3. Some fine-tuning may be needed on the newer platform, but investigating whether the optimization strategies are still relevant for new hardware with higher memory capacity, higher memory bandwidth, and plenty of inter-device communication bandwidth is appealing. Especially comparing the performance of purpose-built AI/ML clusters that provide the highest bandwidth possible to connect every compute node with the performance of general-purpose clusters that are often built with simpler networking topology that is cost-effective.

Inter-node communication is still the key bottleneck of large-scale wave simulations; thus, this topic remains relevant for future research. As an early exploration, the partial ghost exchange, discussed in Section 6.5, is worth exploring for further optimizing the ghost exchange. Another exciting idea is compressed ghost exchange, where the floating-point data is compressed using lossless compression methods. Although this means additional compute overhead, smart network interface cards (smart NICs) can be used to offload data compression. The ideas from prior works by Li et al. (2023, 2024); Lindstrom and Isenburg (2006); Ratanaworabhan et al. (2006) may be applicable for reducing the ghost exchange overhead in wave simulations by compressing the floating-point data. Unlike lossy compression, which uses reduced-precision data, lossless compression should not impact numerical accuracy. In addition, exploring other floating-point formats and their combinations to perform reduced-precision ghost exchange with minimal impact on numerical accuracy is also appealing. For example, 32-bit POSIT promises the same accuracy as the 64-bit IEEE 754 floating-

point format (double precision), as shown by Gustafson and Yonemoto (2017); Buoncristiani et al. (2020); Ciocirlan et al. (2021).

Finally, as an emerging computing technology, there are still a lot of research opportunities with Processing-in-Memory to handle large-scale wave simulations. For example, the multi-PIM chips to handle larger scale wave simulation is not yet explored in this dissertation. This includes implementing a message-passing library for PIM, developing an adaptive parallel mesh refinement library for PIM, communication and synchronization strategies between PIM chips, and optimizing the inter-device data movement. Another research idea is to integrate the newly developed compilers and frameworks to map the wave simulations into PIM and compare their performance against manual code porting. For example, the frameworks proposed in prior works by Ma et al. (2024); Shin et al. (2023); Hadidi et al. (2017); Ahn et al. (2015b) may be helpful to port wave simulation codes into PIM.

Appendix A: Mathematical Review

This appendix provides a brief mathematical review of the acoustic wave equation, elastic wave equation, and discontinuous Galerkin (dG) discretization as basic knowledge to help understand this dissertation. In addition, for simplification, many derivations in this appendix are limited to one- or two-dimensional spaces. However, they are easily extendable to the three-dimensional spaces which is used in this dissertation. For a more thorough explanation, readers are suggested to consult the given references.

A.1 The Acoustic Wave Equation

In acoustic wave problems, two unknown variables are the focus of the computation: pressure (p) and particle velocity ($\mathbf{v}$). Both are functions of space and time and can be written as $p(x, y, t)$ and $\mathbf{v}(x, y, t)$ for the case of two-dimensional space. Equations (A.1) and (A.2) present the compact form of the variables where κ is the bulk modulus that defines the resistance of the material to compression, and ρ is the material density. Both equations are solved in each time step during the simulation.

$$\frac{\partial P}{\partial t} + \kappa \nabla \cdot \mathbf{v} = 0 \tag{A.1}$$

$$\frac{\partial \mathbf{v}}{\partial t} + \frac{1}{\rho} \nabla p = 0 \tag{A.2}$$

Opening the divergence operator (∇) in Equation (A.1) gives Equation (A.3). Similarly, opening the divergence operator for Equation (A.2) in each direction results in Equation (A.4). The wave velocity under a medium can be written as a function of bulk modulus and material density as shown in Equation (A.5). Inside the water, C_p will be around $1,440$ m/s.

$$\frac{\partial p}{\partial t} + \kappa \left(\frac{\partial \mathbf{v}_x}{\partial x} + \frac{\partial \mathbf{v}_y}{\partial x} \right) = 0 \tag{A.3}$$

$$\left. \begin{aligned} \frac{\partial \mathbf{v}_x}{\partial t} + \frac{1}{\rho} \frac{\partial p}{\partial x} = 0 \\ \frac{\partial \mathbf{v}_y}{\partial t} + \frac{1}{\rho} \frac{\partial p}{\partial y} = 0 \end{aligned} \right\} \tag{A.4}$$

$$C_p = \sqrt{\frac{\kappa}{\rho}} \tag{A.5}$$

A.2 The Elastic Wave Equation

In elastic wave problems, two unknown variables are the focus of the computation: stress ($\mathcal{S}$) and particle velocity ($\mathbf{v}$). Equations (A.6) and (A.7) show the compact form of the variables where λ, μ, and ρ describe the material properties. λ and μ are generally referred to as Lamé's first parameter and Lamé's second parameter, while ρ is the density of the material. They are related to compressional velocity C_p and shear velocity C_s as shown in Equations (A.8) and (A.9).

$$\frac{\partial \mathcal{S}}{\partial t} = \mu \left(\nabla \mathbf{v} + \nabla \mathbf{v}^T \right) + \lambda \nabla \cdot \mathbf{v} \, \mathcal{I} \tag{A.6}$$

$$\frac{\partial \mathbf{v}}{\partial t} = \frac{1}{\rho} \nabla \cdot \mathcal{S} \tag{A.7}$$

$$C_p = \sqrt{\frac{\lambda + 2\mu}{\rho}} \tag{A.8}$$

$$C_s = \sqrt{\frac{\mu}{\rho}} \tag{A.9}$$

The stress $\mathcal{S}$ is a matrix (stress tensor) while the particle velocity $\mathbf{v}$ is a vector as shown in Equation (A.10) for two-dimensional space and Equation (A.11) for three-dimensional space.

$$\mathbf{v} = \begin{bmatrix} u \\ v \end{bmatrix} \quad \text{and} \quad \mathcal{S} = \begin{bmatrix} S_{11} & S_{12} \\ S_{21} & S_{22} \end{bmatrix} \tag{A.10}$$

$$\mathbf{v} = \begin{bmatrix} u \\ v \\ w \end{bmatrix} \quad \text{and} \quad \mathcal{S} = \begin{bmatrix} S_{11} & S_{12} & S_{13} \\ S_{21} & S_{22} & S_{23} \\ S_{31} & S_{32} & S_{33} \end{bmatrix} \tag{A.11}$$

A.3 Discontinuous Galerkin (dG) Discretization

A.3.1 Overview of dG

The discontinuous Galerkin (dG) discretization method is used thoroughly in this dissertation, and thus, having some familiarity with dG will be helpful. The dG method is a compact and robust finite element method that is locally conservative, stable, and high-order accurate as described by Cockburn et al. (2000); Grote et al. (2006). The dG method can easily handle complex geometries, approximations of the polynomials in different degrees between different elements, and irregular meshes with hanging nodes. Curious readers are suggested to consult the book by Hesthaven and Warburton (2010) for a more detailed explanation of dG.

As explained in the work by Baggag et al. (2000), the dG method is appropriate for parallel computation since each element can be computed independently. Each element can be considered as an individual entity that needs to obtain some boundary data from its neighboring elements. That is, the (interior) equations are local to each finite element, and thus, the solution can be computed within the element without looking for its neighboring elements. The only time the elements need to talk to each other is during flux computation.

A.3.2 Discretizing Problem Domain

For demonstration purposes, a problem domain Ω in one-dimensional space is defined as shown in Figure A.1, with problem formulation given in Equation (A.12). The f is a known flux function (e.g., $f = au$) while $u(x,t)$ is an unknown function. In acoustic wave (Appendix A.1), the u can be pressure $p(x,t)$ or particle velocity $\mathbf{v}(x,t)$ while in elastic wave (Appendix A.2) u can be stress $\mathcal{S}(x,t)$ or particle velocity $\mathbf{v}(x,t)$.

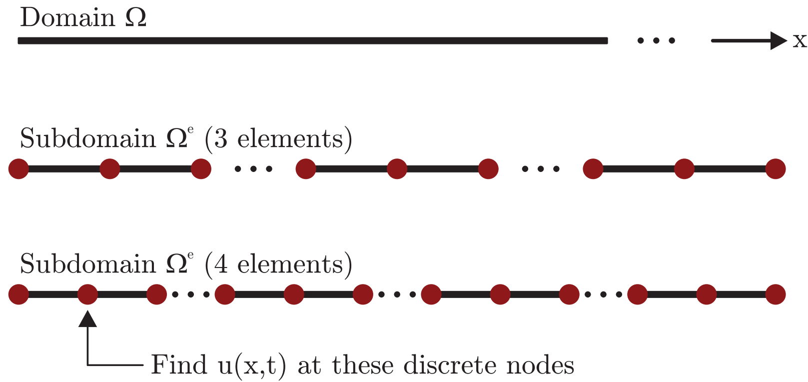


Figure A.1: An example of problem domain for discretization in 1D space.

$$\frac{\partial u}{\partial t} + \frac{\partial f}{\partial x} = 0, x \in \Omega \tag{A.12}$$

Substituting f with au results in Equation (A.13), which is a generalization of Equations (A.1) and (A.2).

$$\frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} = 0, x \in \Omega \tag{A.13}$$

Ideally, the $u(x,t)$ is calculated for every point x inside the problem domain Ω. However, it is impossible to solve $u(x,t)$ for every x because it is computationally expensive, and thus discretization comes into the picture. Instead of solving for every x, $u(x,t)$ is solved for some discrete points in the problem domain. The same

principles apply to the problem domain in two- and three-dimensional space as shown in Figure A.2.

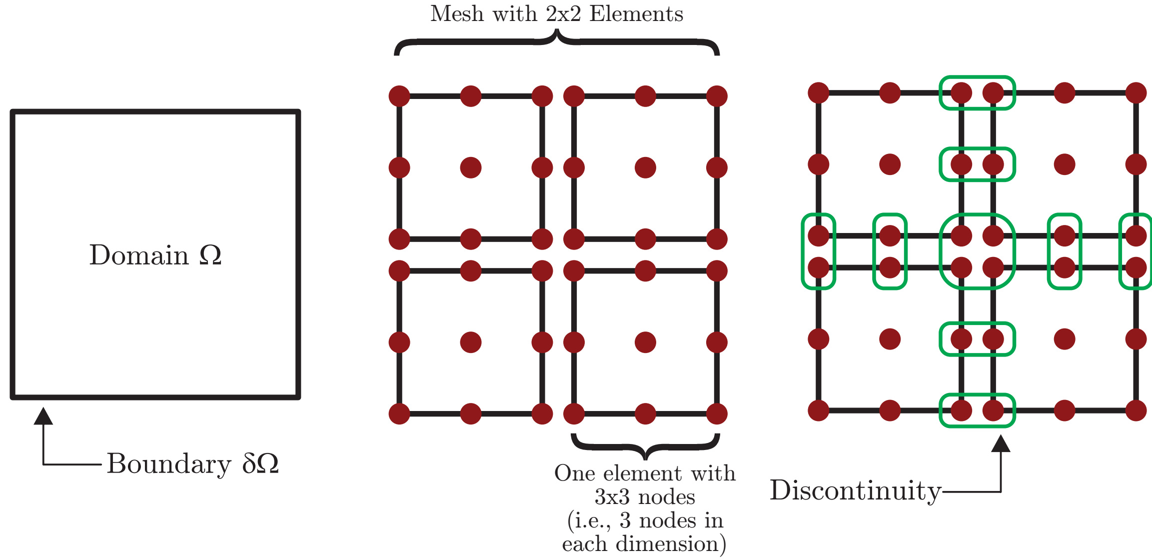


Figure A.2: An example of problem domain for discretization in 2D space showing discontinuity of solution as result of dG discretization.

The result of problem domain discretization is smaller elements that are generally called quadrilateral and hexahedra in two- and three-dimensional spaces, respectively. Each element contains several red dots representing the discrete points where $u(x, t)$, $u(x, y, t)$, and $u(x, y, z, t)$ are evaluated for one-, two-, and three-dimensional spaces, respectively. These discrete points are called observation points or nodes.

A.3.3 Higher Order Element

Each element can have higher number of nodes, also called higher-order, to improve the numerical fidelity at the expense of more computation costs. Higher-order elements are useful for evaluating high-frequency waves since they give sharper images. These high-frequency waves are more susceptible to dispersion as they travel in space, causing them to distort. This phenomenon has been investigated by Poursartip et al. (2020), and the mitigation requires more computing power. The solutions

can either use finer mesh (i.e., dividing the mesh into smaller elements) or increase the order of the elements by adding more nodes. The latter is more economical regarding hardware and numerical perspectives: 1) from the hardware perspective, higher-order elements have higher arithmetic intensity, more local operations, and are more cache-friendly; and 2) from numerical perspective, higher-order elements have higher spectral accuracy. In addition, from a mathematical perspective, using higher-order elements is more efficient as it converges quickly.

A.3.4 Discontinuity of Solutions

The solutions obtained from dG discretization are discontinuous. The right side of Figure A.2 shows the discontinuity of solutions. The nodes circled in green physically touch each other (i.e., located in the same coordinate in space). However, they can have different solutions (i.e., discontinuous due to non single-valued) since each node is evaluated in the respective element in which it is located. In other words, the solutions inside an element are always continuous, while the solutions between elements can be discontinuous. This discontinuity must be addressed, which will be discussed later in Appendix A.3.5.

A.3.5 Deriving Continuous Weak Form

To derive the strong form, Equation (A.12) is multiplied by a test function $\tilde{u}$, which is a nice function (i.e., smooth function), and is integrated over one element Ω^e resulting in Equation (A.14). The transformation from differential form to integral form is done for every element inside the mesh.

$$\int_{\Omega^e} \tilde{u} \left(\frac{\partial u}{\partial t} + \frac{\partial f}{\partial x} \right) dx = 0 \tag{A.14}$$

During the discretization of u and f in Equation (A.12), instead of computing the exact solution of u, the approximate solution u_h over h field is calculated. However, since u_h is the approximate solution to u, it may not satisfy Equation (A.12),

as shown in Equation (A.15).

$$\frac{\partial u_h}{\partial t} + \frac{\partial f_h}{\partial x} \neq 0 \tag{A.15}$$

By multiplying Equation (A.15) with test function $\tilde{u}$ and transforming it to integral form, it can be said that the resulting Equation (A.16) equal to zero. The test function $\tilde{u}$ is a function of space (i.e., $\tilde{u}(x)$, $\tilde{u}(x, y)$, and $\tilde{u}(x, y, z)$ for one-, two-, and three-dimensional spaces).

$$\int_{\Omega^e} \tilde{u} \left(\frac{\partial u_h}{\partial t} + \frac{\partial f_h}{\partial x} \right) dx = \underbrace{\int_{\Omega^e} \tilde{u} \frac{\partial u_h}{\partial t}\, dx}_{\text{temporal term}} + \underbrace{\int_{\Omega^e} \tilde{u} \frac{\partial f_h}{\partial x}\, dx}_{\text{spatial term}} = 0 \tag{A.16}$$

Applying integration by part to the spatial term of Equation (A.16), the derivative is moved from f_h to $\tilde{u}$, as shown in Equation (A.17). This equation has two terms: element boundary and element interior. Figure A.3 shows how the elements touch each other and how the boundary of each element interacts with neighboring elements. As its name suggests, the computation of the element's interior term is performed inside the element while the element boundary term is computed along the element's boundary. The $\underset{\sim}{n}$ is the normal vector perpendicular to the boundary, ds denotes the integration is performed over the boundary of an element, and dx denotes the integration is performed over the interior of an element.

$$\int_{\Omega^e} \tilde{u} \frac{\partial f_h}{\partial x}\, dx = \underbrace{\int_{\partial\Omega^e} \tilde{u} f_h \underset{\sim}{n}\, ds}_{\text{element boundary}} - \underbrace{\int_{\Omega^e} \frac{\partial \tilde{u}}{\partial x} f_h\, dx}_{\text{element interior}} = 0 \tag{A.17}$$

Until this point, the element has not talked to its neighboring elements because the integration is done over each element, both inside the element and at its boundary. Without exchanging information, the solution can be discontinuous at the boundary, as discussed in Appendix A.3.4. To allow the element to talk to its

neighboring elements, a numerical flux f_h^* is introduced. It is responsible for conveying the information from one element to its neighboring elements to solve the discontinuity problem. A simple example of numerical flux for $f = au$ is given in Equation (A.18), where it averages the solutions obtained from all adjacent elements for the node occupying the same physical space.

$$f_h^* = \frac{1}{2}\left((au)_{left} + (au)_{right}\right) \tag{A.18}$$

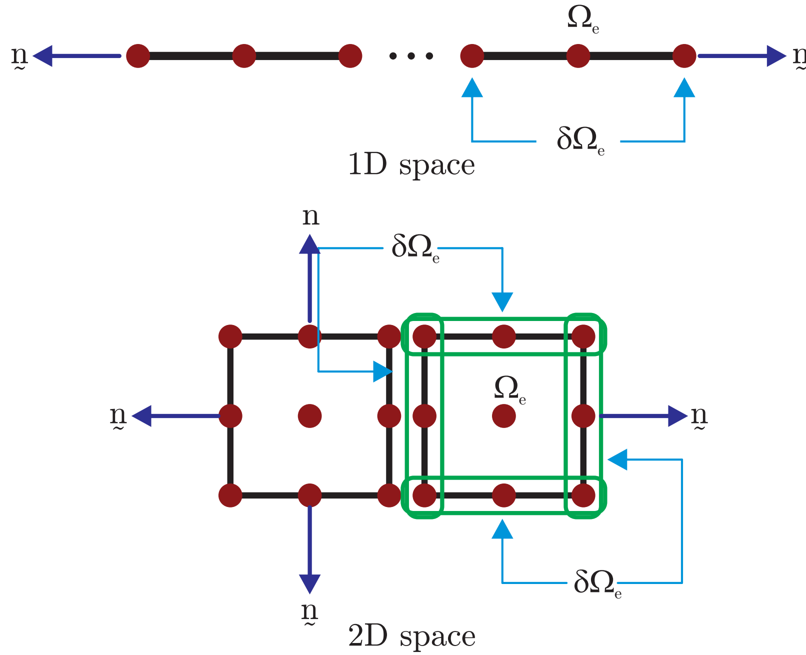


Figure A.3: Boundary of an element, highlighted in green, with vector normal to the boundary, drawn in dark blue lines.

Since numerical flux is not unique, different people can propose different numerical fluxes. Substituting this numerical flux into Equation (A.17) yields Equation (A.19).

$$\int_{\Omega^e} \tilde{u}\frac{\partial f_h}{\partial x}\,dx = \underbrace{\int_{\partial\Omega^e} \tilde{u} f_h^* \underset{\sim}{n}\,ds}_{\text{require neighbor element}} - \underbrace{\int_{\Omega^e} \frac{\partial \tilde{u}}{\partial x} f_h\,dx}_{\text{internal to element}} = 0 \tag{A.19}$$

Next, partial integration is performed to the internal term of Equation (A.19) to move the derivative into f_h as shown in Equation (A.20).

$$\int_{\Omega^e} \frac{\partial \tilde{u}}{\partial x} f_h \, dx = \int_{\partial\Omega^e} \tilde{u} f_h \underset{\sim}{n} \, ds - \int_{\Omega^e} \tilde{u} \frac{\partial f_h}{\partial x} \, dx = 0 \tag{A.20}$$

Substituting the internal term in Equation (A.19) using Equation (A.20) yields Equation (A.21). Further simplification of Equation (A.21) yields Equation (A.22).

$$\int_{\Omega^e} \tilde{u} \frac{\partial f_h}{\partial x} \, dx = \int_{\partial\Omega^e} \tilde{u} f_h^* \underset{\sim}{n} \, ds - \left(\int_{\partial\Omega^e} \tilde{u} f_h \underset{\sim}{n} \, ds - \int_{\Omega^e} \tilde{u} \frac{\partial f_h}{\partial x} \, dx \right) \tag{A.21}$$

$$\int_{\Omega^e} \tilde{u} \frac{\partial f_h}{\partial x} \, dx = \int_{\partial\Omega^e} \tilde{u} \left(f_h^* \underset{\sim}{n} - f_h \underset{\sim}{n} \right) ds + \int_{\Omega^e} \tilde{u} \frac{\partial f_h}{\partial x} \, dx \tag{A.22}$$

By substituting the spatial term of Equation (A.16) with Equation (A.22), final continuous weak-form is obtained as shown in Equation (A.23), which will be discretized.

$$\int_{\Omega^e} \tilde{u} \frac{\partial u_h}{\partial t} \, dx + \int_{\partial\Omega^e} \tilde{u} \left(f_h^* \underset{\sim}{n} - f_h \underset{\sim}{n} \right) ds + \int_{\Omega^e} \tilde{u} \frac{\partial f_h}{\partial x} \, dx = 0 \tag{A.23}$$

A.3.6 Discretization in Space

The next step is to represent the continuous weak form shown in Equation (A.23), which is in integral form, with polynomial approximation. In other words, this transformation changes integration to linear algebra, which will be performed using matrix multiplication. Suppose there is an unknown function $u(x)$ in one-dimensional space as shown in Figure A.4. Shape function $\Psi(x)$, shown in Equation (A.24) can be used to approximate $u(x)$. In this case, $u(x_1) = u_1$ since $\Psi_1(x_1) = 1$, $\Psi_2(x_1) = 0$, and $\Psi_3(x_1) = 0$.

$$u(x) = \Psi_1(x)u_1 + \Psi_2(x)u_2 + \Psi_3(x)u_3 \tag{A.24}$$

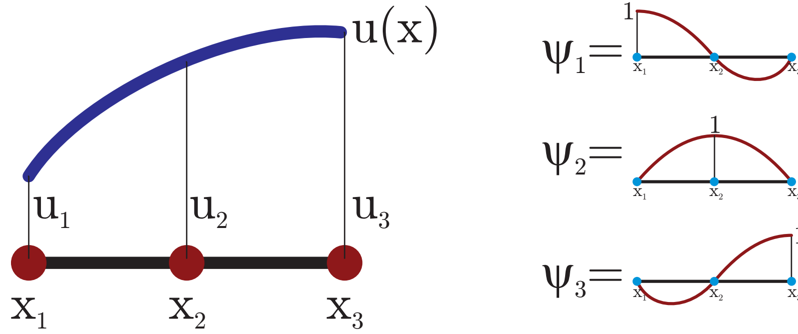


Figure A.4: Example of discretization in one-dimensional space and approximation using polynomial.

The shape function in Equation (A.24) can be written in matrix notation as shown in Equation (A.25). Further, this equation can be written as Equation (A.26), which is called polynomial interpolation where $\underset{\sim}{u}^e$ is inside an element. Remember that the shape function $\Psi(x)$ is already known, and thus the solution is calculated only for the unknown $u(x)$. In addition, the number of shape functions depends on the number of nodes in each dimension; eight nodes in one dimension (e.g., x dimension) means eight shape functions are needed.

$$u(x) = \begin{bmatrix} \Psi_1(x) & \Psi_2(x) & \Psi_3(x) \end{bmatrix}^{\mathsf{T}} \cdot \underbrace{\begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix}}_{\text{constant}} \tag{A.25}$$

$$u(x) = \underset{\sim}{N}^{\mathsf{T}} \underset{\sim}{u}^e \tag{A.26}$$

With the shape functions defined, u and $\tilde{u}$ can be written as polynomial interpolation as shown in Equations (A.27) to (A.30). Note that $\underset{\sim}{N}_x^{\mathsf{T}}$ is the derivative of shape functions.

$$u(x) = \underset{\sim}{N}^{\mathsf{T}} \underset{\sim}{u}^e \tag{A.27}$$

$$\tilde{u}(x) = \underset{\sim}{N}^{\mathsf{T}} \tilde{\underset{\sim}{u}}^e = \left(\tilde{\underset{\sim}{u}}^e\right)^{\mathsf{T}} \underset{\sim}{N} \tag{A.28}$$

$$\frac{\partial u}{\partial x} = \frac{\partial \Psi_1(x)}{\partial x} u_1 + \frac{\partial \Psi_2(x)}{\partial x} u_2 + \frac{\partial \Psi_3(x)}{\partial x} u_3 = \underset{\sim}{N}_x^{\mathsf{T}} \underset{\sim}{u}^e \tag{A.29}$$

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial t}\left(\underset{\sim}{N}^{\mathsf{T}} \underset{\sim}{u}^e\right) = \underset{\sim}{N}^{\mathsf{T}} \frac{\partial \underset{\sim}{u}^e}{\partial t} = \underset{\sim}{N}^{\mathsf{T}} \dot{\underset{\sim}{u}}^e \tag{A.30}$$

The ingredients to discretize the final continuous weak form in Equation (A.23) have been prepared. First, Equation (A.23) is rewritten in Equation (A.31). Secondly, the time term in Equation (A.31) is manipulated by substituting $\tilde{u}$ and $\frac{\partial u_h}{\partial t}$ with polynomial approximation shown in Equation (A.32). A mass matrix $\underset{\sim}{M}$ is obtained in Equation (A.33).

$$\underbrace{\int_{\Omega^e} \tilde{u} \frac{\partial u_h}{\partial t}\, dx}_{\text{time term}} + \underbrace{\int_{\partial\Omega^e} \tilde{u}\left(f_h^* \underset{\sim}{n} - f_h \underset{\sim}{n}\right) ds}_{\text{flux contributions term}} + \underbrace{\int_{\Omega^e} \tilde{u} \frac{\partial f_h}{\partial x}\, dx}_{\text{volume contributions term}} = 0 \tag{A.31}$$

$$\int_{\Omega^e} \tilde{u} \frac{\partial u_h}{\partial t}\, dx = \int_{\Omega^e} \left(\tilde{\underset{\sim}{u}}^e\right)^{\mathsf{T}} \underset{\sim}{N} \underset{\sim}{N}^{\mathsf{T}} \dot{\underset{\sim}{u}}^e\, dx = \left(\tilde{\underset{\sim}{u}}^e\right)^{\mathsf{T}} \underbrace{\left(\int_{\Omega^e} \underset{\sim}{N} \underset{\sim}{N}^{\mathsf{T}}\, dx\right)}_{\text{mass matrix}} \dot{\underset{\sim}{u}}^e \tag{A.32}$$

$$\underset{\sim}{M} = \int_{\Omega^e} \underset{\sim}{N} \underset{\sim}{N}^{\mathsf{T}}\, dx \tag{A.33}$$

Just like in Equations (A.25) and (A.26), the mass matrix shown in Equation (A.33) can be rewritten into Equation (A.34), which is further written as Equation (A.35) by opening the matrix multiplication.

$$\underset{\sim}{M} = \int_{\Omega^e} \begin{bmatrix} \Psi_1(x) \\ \Psi_2(x) \\ \Psi_3(x) \end{bmatrix} \cdot \begin{bmatrix} \Psi_1(x) & \Psi_2(x) & \Psi_3(x) \end{bmatrix} dx \tag{A.34}$$

$$\underset{\sim}{M} = \int_{\Omega^e} \begin{bmatrix} \Psi_1(x)\Psi_1(x) & \Psi_1(x)\Psi_2(x) & \Psi_1(x)\Psi_3(x) \\ \Psi_2(x)\Psi_1(x) & \Psi_2(x)\Psi_2(x) & \Psi_2(x)\Psi_3(x) \\ \Psi_3(x)\Psi_1(x) & \Psi_3(x)\Psi_2(x) & \Psi_3(x)\Psi_3(x) \end{bmatrix} dx \tag{A.35}$$

With a special integration scheme called GLL (Gauss-Lobatto-Legendre) as described by Parter (1999); Winckel (2024), the matrix in Equation (A.35) becomes a diagonal matrix if the integration points are chosen carefully. Computing the inverse of the mass matrix is a very cheap operation if it is a diagonal matrix.

The third step is to manipulate the volume contributions term of Equation (A.31) using $f_h = au$. With the substitution using Equations (A.27) to (A.30), Equation (A.36) is obtained, which is further refined in Equation (A.37) by rearranging the terms. The stiffness matrix $\underset{\sim}{K}$ is given in Equation (A.38).

$$\int_{\Omega^e} \tilde{u} \frac{\partial f_h}{\partial x}\, dx = \int_{\Omega^e} \frac{\partial \tilde{u}}{\partial x} au\, dx = \int_{\Omega^e} (\underset{\sim}{\tilde{u}}^e)^\mathsf{T}\, \underset{\sim}{N} a \underset{\sim}{N}_x^\mathsf{T} \underset{\sim}{u}^e\, dx \tag{A.36}$$

$$\int_{\Omega^e} \frac{\partial \tilde{u}}{\partial x} au\, dx = \int_{\Omega^e} (\underset{\sim}{\tilde{u}}^e)^\mathsf{T}\, \underset{\sim}{N} a \underset{\sim}{N}_x^\mathsf{T} \underset{\sim}{u}^e\, dx = (\underset{\sim}{\tilde{u}}^e)^\mathsf{T} \underbrace{\left(\int_{\Omega^e} a \underset{\sim}{N} \underset{\sim}{N}_x^\mathsf{T}\, dx \right)}_{\text{stiffness matrix}} \underset{\sim}{u}^e \tag{A.37}$$

$$\underset{\sim}{K} = \int_{\Omega^e} a \underset{\sim}{N} \underset{\sim}{N}_x^\mathsf{T}\, dx \tag{A.38}$$

The fourth step is to manipulate the flux contributions term of Equation (A.31), resulting in Equation (A.39). The flux contributions $\underset{\sim}{F}$ is given in Equation (A.40).

$$\int_{\partial\Omega^e} \tilde{u} \left(f_h^* \underset{\sim}{n} - f_h \underset{\sim}{n} \right) ds = (\underset{\sim}{\tilde{u}}^e)^\mathsf{T} \underbrace{\int_{\partial\Omega^e} \underset{\sim}{N} \underset{\sim}{n} \left(f^* - f \right) ds}_{\text{flux contribution}} \tag{A.39}$$

$$\underset{\sim}{F} = \int_{\partial\Omega^e} \underset{\sim}{N} \underset{\sim}{n} \left(f^* - f \right) ds \tag{A.40}$$

Combining Equations (A.32), (A.37), and (A.39) yields Equation (A.41). Since the test function $\tilde{u}$ is non-zero and can have arbitrary values, it can be removed from Equation (A.41) to form Equation (A.42).

$$(\tilde{\underset{\sim}{u}}^e)^\mathsf{T} \left(\int_{\Omega^e} \underset{\sim}{N} \underset{\sim}{N}^\mathsf{T} \, dx \right) \dot{\underset{\sim}{u}}^e + (\tilde{\underset{\sim}{u}}^e)^\mathsf{T} \left(\int_{\Omega^e} a \underset{\sim}{N} \underset{\sim}{N}_x^\mathsf{T} \, dx \right) \underset{\sim}{u}^e + (\tilde{\underset{\sim}{u}}^e)^\mathsf{T} \int_{\partial\Omega^e} \underset{\sim}{N} \underset{\sim}{n} \, (f^* - f) \, ds = 0 \tag{A.41}$$

$$\left(\int_{\Omega^e} \underset{\sim}{N} \underset{\sim}{N}^\mathsf{T} \, dx \right) \dot{\underset{\sim}{u}}^e + \left(\int_{\Omega^e} a \underset{\sim}{N} \underset{\sim}{N}_x^\mathsf{T} \, dx \right) \underset{\sim}{u}^e + \int_{\partial\Omega^e} \underset{\sim}{N} \underset{\sim}{n} \, (f^* - f) \, ds = 0 \tag{A.42}$$

Substituting Equations (A.33), (A.38), and (A.40) into Equation (A.42) yields Equation (A.43) where $\underset{\sim}{C}$ is the contributions from volume and flux.

$$\underset{\sim}{M} \dot{\underset{\sim}{u}}^e + \underbrace{\underset{\sim}{K} \underset{\sim}{u}^e + \underset{\sim}{F}}_{\underset{\sim}{C}} = 0 \tag{A.43}$$

A.3.7 Discretization in Time

Equation (A.43) can be rewritten as Equation (A.44). The time-stepping with the time step of Δt is done as shown in Equation (A.45).

$$\underset{\sim}{M} \dot{\underset{\sim}{u}}^e + \underset{\sim}{C} = 0 \tag{A.44}$$

$$\underset{\sim}{M} \frac{\dot{\underset{\sim}{u}}^{e(n+1)} - \dot{\underset{\sim}{u}}^{e(n)}}{\Delta t} + \underset{\sim}{C}^n = 0 \tag{A.45}$$

Rearranging the Equation (A.45) yields Equation (A.46). The $\underset{\sim}{M}^{-1}$ denotes the inverse mass matrix, which is a diagonal matrix.

$$\dot{\underset{\sim}{u}}^{e(n+1)} = \dot{\underset{\sim}{u}}^{e(n)} - \Delta t \underset{\sim}{M}^{-1} \underset{\sim}{C}^n \tag{A.46}$$

A.4 Example of Discontinuous Galerkin (DG) in Acoustic Wave

This section provides a brief explanation of how to apply dG discretization (Appendix A.3) into acoustic waves equation (Appendix A.1). Although this section uses two-dimensional space to simplify writing, extension to three-dimensional space should be straightforward. Equations (A.3) and (A.4) are rewritten into Equations (A.47) and (A.48), respectively, by replacing the derivative of pressure p and $\mathbf{v}$ with respect to time as $\dot{p}$ and $\dot{\mathbf{v}}$, respectively.

$$\dot{p} + \kappa(x)\nabla \underset{\sim}{\mathbf{v}} = 0 \tag{A.47}$$

$$\dot{\underset{\sim}{\mathbf{v}}} + \frac{1}{\rho(x)}\nabla p = 0 \tag{A.48}$$

Applying test function $\tilde{p}$ and $\tilde{\underset{\sim}{\mathbf{v}}}$ to Equations (A.47) and (A.48) yields Equations (A.49) and (A.50).

$$\int_{\Omega^e} \tilde{p}\left(\dot{p} + \kappa(x)\nabla \underset{\sim}{\mathbf{v}}\right) dx = 0 \tag{A.49}$$

$$\int_{\Omega^e} \tilde{\underset{\sim}{\mathbf{v}}} \left(\dot{\underset{\sim}{\mathbf{v}}} + \frac{1}{\rho(x)}\nabla p\right) dx = 0 \tag{A.50}$$

A.4.1 Deriving The Continuous Weak Form

Equation (A.49) is further manipulated using the same steps as going from Equation (A.16) to Equation (A.22). The step-by-step is shown in Equations (A.51) to (A.53), resulting in Equation (A.54).

$$\int_{\Omega^e} \tilde{p}\kappa\nabla \underset{\sim}{\mathbf{v}}\, dx = \int_{\partial\Omega^e} \tilde{p}\kappa \underset{\sim}{\mathbf{v}} \underset{\sim}{n}\, ds - \int_{\Omega^e} \kappa\nabla\tilde{p}\underset{\sim}{\mathbf{v}}\, dx \tag{A.51}$$

$$\int_{\Omega^e} \tilde{p}\kappa\nabla\underset{\sim}{\mathbf{v}}\,dx = \int_{\partial\Omega^e} \tilde{p}\kappa \underbrace{\left(\underset{\sim}{\mathbf{v}}\underset{\sim}{n}\right)^*}_{\substack{\text{numerical}\\ \text{flux}}} ds - \int_{\Omega^e} \kappa\nabla\tilde{p}\underset{\sim}{\mathbf{v}}\,dx \tag{A.52}$$

$$\int_{\Omega^e} \tilde{p}\kappa\nabla\underset{\sim}{\mathbf{v}}\,dx = \int_{\partial\Omega^e} \tilde{p}\kappa\left(\underset{\sim}{\mathbf{v}}\underset{\sim}{n}\right)^* ds - \int_{\partial\Omega^e} \tilde{p}\kappa\underset{\sim}{\mathbf{v}}\underset{\sim}{n}\,ds + \int_{\Omega^e} \kappa\tilde{p}\nabla\underset{\sim}{\mathbf{v}}\,dx \tag{A.53}$$

$$\int_{\Omega^e} \tilde{p}\kappa\nabla\underset{\sim}{\mathbf{v}}\,dx = \int_{\partial\Omega^e} \tilde{p}\left(\kappa\left(\underset{\sim}{\mathbf{v}}\underset{\sim}{n}\right)^* - \kappa\left(\underset{\sim}{\mathbf{v}}\underset{\sim}{n}\right)\right) ds + \int_{\Omega^e} \kappa\tilde{p}\nabla\underset{\sim}{\mathbf{v}}\,dx \tag{A.54}$$

The same manipulation is done to Equation (A.50), resulting in Equation (A.55) with result shown in Equation (A.56). Note that some steps are not shown.

$$\int_{\Omega^e} \underset{\sim}{\tilde{\mathbf{v}}}\frac{1}{\rho}\nabla p\,dx = \int_{\partial\Omega^e} \frac{1}{\rho}p\underset{\sim}{\tilde{\mathbf{v}}}\underset{\sim}{n}\,ds - \int_{\Omega^e} \frac{1}{\rho}\underset{\sim}{\tilde{\mathbf{v}}}\nabla p\,dx \tag{A.55}$$

$$\int_{\Omega^e} \underset{\sim}{\tilde{\mathbf{v}}}\frac{1}{\rho}\nabla p\,dx = \int_{\partial\Omega^e} \underset{\sim}{\tilde{\mathbf{v}}}\left(\frac{1}{\rho}\left(p\underset{\sim}{n}\right)^* - \frac{1}{\rho}\left(p\underset{\sim}{n}\right)\right) ds + \int_{\Omega^e} \underset{\sim}{\tilde{\mathbf{v}}}\frac{1}{\rho}\nabla p\,dx \tag{A.56}$$

Finally, complete continuous weak-form can be written by combining the time term, flux contributions term, and volume contributions term as shown in Equations (A.57) and (A.58).

$$\int_{\Omega^e} \tilde{p}\dot{p}\,dx + \int_{\Omega^e} \kappa\tilde{p}\nabla\underset{\sim}{\mathbf{v}}\,dx + \int_{\partial\Omega^e} \tilde{p}\left(\kappa\left(\underset{\sim}{\mathbf{v}}\underset{\sim}{n}\right)^* - \kappa\left(\underset{\sim}{\mathbf{v}}\underset{\sim}{n}\right)\right) ds = 0 \tag{A.57}$$

$$\int_{\Omega^e} \underset{\sim}{\tilde{\mathbf{v}}}\underset{\sim}{\dot{\mathbf{v}}}\,dx + \int_{\Omega^e} \underset{\sim}{\tilde{\mathbf{v}}}\frac{1}{\rho}\nabla p\,dx + \int_{\partial\Omega^e} \underset{\sim}{\tilde{\mathbf{v}}}\left(\frac{1}{\rho}\left(p\underset{\sim}{n}\right)^* - \frac{1}{\rho}\left(p\underset{\sim}{n}\right)\right) ds = 0 \tag{A.58}$$

A.4.2 Discretization in Space

Following the same step as in Appendix A.3.6, shape functions in two-dimensional space are defined in Equation (A.59). Then, the polynomial approximation is performed by defining several equations as shown in Equations (A.60) to (A.62).

$$p(x,y) = p_1\Psi_1(x,y) + p_2\Psi_2(x,y) + \ldots + p_n\Psi_n(x,y) = \underset{\sim}{N}^{\mathsf{T}}\underset{\sim}{p}^e \tag{A.59}$$

$$\frac{\partial p}{\partial x} = \frac{\partial \Psi_1}{\partial x}p_1 + \ldots + \frac{\partial \Psi_n}{\partial x}p_n = \underset{\sim}{N}_x^{\mathsf{T}}\underset{\sim}{p}^e \tag{A.60}$$

$$\frac{\partial p}{\partial y} = \frac{\partial \Psi_1}{\partial y}p_1 + \ldots + \frac{\partial \Psi_n}{\partial y}p_n = \underset{\sim}{N}_y^{\mathsf{T}}\underset{\sim}{p}^e \tag{A.61}$$

$$\underset{\sim}{\mathbf{v}} \begin{cases} v_1 = \underset{\sim}{N}^{\mathsf{T}}\underset{\sim}{\mathbf{v}}_1^e \\ v_2 = \underset{\sim}{N}^{\mathsf{T}}\underset{\sim}{\mathbf{v}}_2^e \end{cases} \tag{A.62}$$

Before going into the substitution, three terms are defined in Equations (A.63) to (A.65).

$$\tilde{p}\nabla\underset{\sim}{\mathbf{v}} = \tilde{p}\left(\frac{\partial v_1}{\partial x} + \frac{\partial v_2}{\partial x}\right) = \left(\underset{\sim}{p}^e\right)^{\mathsf{T}} N \underbrace{\left(\underset{\sim}{N}_x^{\mathsf{T}}\underset{\sim}{\mathbf{v}}_1^e + \underset{\sim}{N}_y^{\mathsf{T}}\underset{\sim}{\mathbf{v}}_2^e\right)}_{div\,\mathbf{v}} \tag{A.63}$$

$$\nabla P = \begin{bmatrix} \frac{\partial p}{\partial x} \\ \frac{\partial p}{\partial y} \end{bmatrix} = \underbrace{\begin{bmatrix} \underset{\sim}{N}_x^{\mathsf{T}}\underset{\sim}{p}^e \\ \underset{\sim}{N}_y^{\mathsf{T}}\underset{\sim}{p}^e \end{bmatrix}}_{grad\,p} \tag{A.64}$$

$$\tilde{\underset{\sim}{\mathbf{v}}}\nabla p = \begin{bmatrix} \tilde{\mathbf{v}}_1 & \tilde{\mathbf{v}}_2 \end{bmatrix} \cdot \begin{bmatrix} \underset{\sim}{N}_x^{\mathsf{T}}\underset{\sim}{p}^e \\ \underset{\sim}{N}_y^{\mathsf{T}}\underset{\sim}{p}^e \end{bmatrix} = \begin{bmatrix} (\tilde{\underset{\sim}{\mathbf{v}}}_1^e)^{\mathsf{T}}\underset{\sim}{N} & (\tilde{\underset{\sim}{\mathbf{v}}}_2^e)^{\mathsf{T}}\underset{\sim}{N} \end{bmatrix} \cdot \begin{bmatrix} \underset{\sim}{N}_x^{\mathsf{T}}\underset{\sim}{p}^e \\ \underset{\sim}{N}_y^{\mathsf{T}}\underset{\sim}{p}^e \end{bmatrix} \tag{A.65}$$

To simplify Equation (A.65), the following definitions are used: $grad\,p[0] = \underset{\sim}{N}_x^{\mathsf{T}}\underset{\sim}{p}^e$ and $grad\,p[1] = \underset{\sim}{N}_y^{\mathsf{T}}\underset{\sim}{p}^e$, leading to Equation (A.66).

$$\tilde{\underset{\sim}{\mathbf{v}}} \nabla p = \left[(\tilde{\underset{\sim}{\mathbf{v}}}_1^e)^\mathsf{T} \quad (\tilde{\underset{\sim}{\mathbf{v}}}_2^e)^\mathsf{T} \right] \cdot \begin{bmatrix} \underset{\sim}{N} * grad\, p[0] \\ \underset{\sim}{N} * grad\, p[1] \end{bmatrix} = (\underset{\sim}{\mathbf{v}}^e)^\mathsf{T} \cdot \begin{bmatrix} \underset{\sim}{N} * grad\, p[0] \\ \underset{\sim}{N} * grad\, p[1] \end{bmatrix} \tag{A.66}$$

With these definitions, Equation (A.57) can be written into Equation (A.67). Then, substituting Equations (A.60) to (A.65) into Equation (A.67) yields Equation (A.68). The test function $\left(\tilde{\underset{\sim}{p}}^e\right)^\mathsf{T}$ can be eliminated, which results in Equation (A.69).

$$\int_{\Omega^e} \tilde{p}\dot{p}\, dx + \int_{\Omega^e} \kappa \tilde{p} \nabla \underset{\sim}{\mathbf{v}}\, dx + \int_{\partial\Omega^e} \tilde{p}\, (\ldots)\, ds = 0 \tag{A.67}$$

$$\int_{\Omega^e} \left(\tilde{\underset{\sim}{p}}^e\right)^\mathsf{T} \underset{\sim}{N}\,\underset{\sim}{N}^\mathsf{T} \dot{\underset{\sim}{p}}^e\, dx + \int_{\Omega^e} \left(\tilde{\underset{\sim}{p}}^e\right)^\mathsf{T} \underset{\sim}{N} * div\, \mathbf{v} * \kappa\, dx + \int_{\partial\Omega^e} \left(\tilde{\underset{\sim}{p}}^e\right)^\mathsf{T} \underset{\sim}{N}\, (\ldots)\, ds = 0 \tag{A.68}$$

$$\int_{\Omega^e} \underset{\sim}{N}\,\underset{\sim}{N}^\mathsf{T} \dot{\underset{\sim}{p}}^e\, dx + \underbrace{\int_{\Omega^e} \underset{\sim}{N} * div\, \mathbf{v} * \kappa\, dx}_{\text{volume term}} + \underbrace{\int_{\partial\Omega^e} \underset{\sim}{N}\, (\ldots)\, ds}_{\text{flux term}} = 0 \tag{A.69}$$

The same steps are done for Equation (A.58), which yields Equation (A.70).

$$\int_{\Omega^e} \tilde{\underset{\sim}{\mathbf{v}}} \dot{\underset{\sim}{\mathbf{v}}}\, dx + \int_{\Omega^e} \tilde{\underset{\sim}{\mathbf{v}}} \frac{1}{\rho} \nabla p\, dx + \int_{\partial\Omega^e} \tilde{\underset{\sim}{\mathbf{v}}}\, (\ldots)\, ds = 0 \tag{A.70}$$

The substituent for $\tilde{\underset{\sim}{\mathbf{v}}} \dot{\underset{\sim}{\mathbf{v}}}^\mathsf{T}$ (i.e., inside time term of Equation (A.70)) is defined in Equation (A.71) and is continued to Equation (A.72).

$$\tilde{\underset{\sim}{\mathbf{v}}} \dot{\underset{\sim}{\mathbf{v}}}^\mathsf{T} = \tilde{\mathbf{v}}_1 \dot{\mathbf{v}}_1 + \tilde{\mathbf{v}}_2 \dot{\mathbf{v}}_2 = (\tilde{\underset{\sim}{\mathbf{v}}}_1^e)^\mathsf{T} \underset{\sim}{N}\,\underset{\sim}{N}^\mathsf{T} \dot{\underset{\sim}{\mathbf{v}}}_1^e + (\tilde{\underset{\sim}{\mathbf{v}}}_2^e)^\mathsf{T} \underset{\sim}{N}\,\underset{\sim}{N}^\mathsf{T} \dot{\underset{\sim}{\mathbf{v}}}_2^e \tag{A.71}$$

$$\tilde{\underset{\sim}{\mathbf{v}}} \dot{\underset{\sim}{\mathbf{v}}}^\mathsf{T} = \left[(\tilde{\underset{\sim}{\mathbf{v}}}_1^e)^\mathsf{T} \quad (\tilde{\underset{\sim}{\mathbf{v}}}_2^e)^\mathsf{T} \right] \cdot \begin{bmatrix} \underset{\sim}{N}\,\underset{\sim}{N}^\mathsf{T} \dot{\underset{\sim}{\mathbf{v}}}_1^e \\ \underset{\sim}{N}\,\underset{\sim}{N}^\mathsf{T} \dot{\underset{\sim}{\mathbf{v}}}_2^e \end{bmatrix} = (\underset{\sim}{\mathbf{v}}^e)^\mathsf{T} \cdot \begin{bmatrix} \underset{\sim}{M} \dot{\underset{\sim}{\mathbf{v}}}_1^e \\ \underset{\sim}{M} \dot{\underset{\sim}{\mathbf{v}}}_2^e \end{bmatrix} \tag{A.72}$$

Finally, Equations (A.66) and (A.72) are used to substitute the time term and volume term of Equation (A.70), respectively, resulting in Equation (A.73).

$$\int_{\Omega^e} \begin{bmatrix} \underset{\sim}{N}\underset{\sim}{N}^{\mathsf{T}}\dot{\underset{\sim}{\mathbf{v}}}_1^e \\ \\ \underset{\sim}{N}\underset{\sim}{N}^{\mathsf{T}}\dot{\underset{\sim}{\mathbf{v}}}_2^e \end{bmatrix} dx + \int_{\Omega^e} \frac{1}{\rho} \begin{bmatrix} \underset{\sim}{N} * grad\, p[0] \\ \\ \underset{\sim}{N} * grad\, p[1] \end{bmatrix} dx + \int_{\partial\Omega^e} \begin{bmatrix} \underset{\sim}{N}\,(\dots) \\ \underset{\sim}{N}\,(\dots) \end{bmatrix} ds = 0 \tag{A.73}$$

A.4.3 Discretization in Time

Following the same steps as in Appendix A.3.7, the semi-discretized equations for both pressure and particle velocity are shown in Equations (A.74) and (A.75). Finally, Equations (A.45) and (A.46) can be used for the time-stepping.

$$\underset{\sim}{M}\dot{\underset{\sim}{p}}^e + \underset{\sim}{C}_p = 0 \tag{A.74}$$

$$\left.\begin{aligned} \underset{\sim}{M}\dot{\underset{\sim}{\mathbf{v}}}_1^e + \underset{\sim}{C}_{v_1} = 0 \\ \underset{\sim}{M}\dot{\underset{\sim}{\mathbf{v}}}_2^e + \underset{\sim}{C}_{v_2} = 0 \end{aligned}\right\} \tag{A.75}$$

Appendix B: Detailed Explanation of Wave Simulation Codes

This appendix[1] provides detailed information on the CPU codes explained in Chapter 4. It gives more detail on the inside of the Element Data Structure (Appendix B.1), the output of the simulation application (Appendix B.2), the execution flow of the main program (Appendix B.3), the execution flow of the simulation kernels, and the compile-time and runtime configurations (Appendices B.5 and B.6). Some diagrams and explanations are adopted with modifications from my Master's Thesis, Hanindhito (2020), which is an early study of this dissertation.

B.1 Element Data Structure

This section of the appendix explains more detail about `ElementDataBase`, described briefly in Section 4.2.3. The `ElementDataBase` is a struct whose contents are explained in Table B.1. All of the struct members are arrays of `real` data type, except for `is_element_curvilinear` and `jacobian_det_domain`, each consists of one value. The `real` data type depends on the precision with which the simulation is compiled, and hence, it is a compile-time configuration (Appendix B.5). For double and single precision simulation, the `real` will be `double` and `float`, respectively.

The number of items for array members depends on the compile-time configurations (Appendix B.5), which are explained as follows.

- `DIMENSION` depends on the dimensionality of the problem space (i.e., 2 and 3

[1]Some materials on this appendix are adapted with modification from my Master's Thesis: Bagus Hanindhito. GPU-accelerated high-performance computing for architecture-aware wave simulation based on discontinuous Galerkin algorithms. UT Electronic Theses and Dissertations, 2020. http://dx.doi.org/10.26153/tsw/42690.
My Master's Thesis only considers acoustic wave simulations, and thus, the modifications performed to make the explanation more general and applicable to elastic wave simulations.

for 2D and 3D spaces, respectively), which is derived from the compile-time configuration `-DDIMENSION`.

- `NNODE` is the number of nodes inside each element (Section 4.1.2), which is derived from the compile-time configurations `-DNNODE_1D` and `-DDIMENSION`.

- `NUM_VARS` is the number of unknown values computed during the simulation, four for acoustic wave simulation and nine for elastic wave simulation in 3D space (Section 2.2). It is automatically derived from the compile-time configuration `-DPROBLEM_TYPE`.

- `NUM_AUX` is the number of auxiliary (temporary) variables for time integrator (Section 2.3.3). Only fourth-order Low-Storage Runge-Kutta Integrator (LSRK4) is available, which only needs one auxiliary variable per variable per node.

- `NUM_MATERIALS` is the number of material properties used during the simulation, two for acoustic wave simulation (κ and ρ) and three for elastic wave simulation (λ, μ, and ρ), as discussed in Section 2.2. It is automatically derived from the compile-time configuration `-DPROBLEM_TYPE`.

- `NNODE_MATERIALS` is the number of material nodes, which is derived from the compile-time configurations `-DNNODE_MATERIAL_1D` and `-DDIMENSION`.

The `xt` stores the space coordinate (x, y, z) of each node inside an element, which depends on the number of nodes in each element (`NNODE`) and the problem dimension. On the other hand, the `is_element_curvilinear` indicates whether the element is rectangular or curvilinear. This dissertation focuses only on rectangular elements. Furthermore, the `jacobian_det_domain`, `jacobian_inverse_domain`, and `jacobian_det_boundary` store the determinant jacobian, diagonal matrix, and determinant jacobian at each face for structured grids, respectively.

Member Name	Number of Items	Data Format	Description
`xt`	`DIMENSION` $\times$`NNODE`	Real	The space coordinate (x, y, z) of each node within an element.
`is_element_curvilinear`	1	Boolean	Indicates whether the element is rectangular or curvilinear.
`jacobian_det_domain`	1	Real	Determinant jacobian for structured grids.
`jacobian_inverse_domain`	`DIMENSION`	Real	Diagonal matrix for structured grids.
`jacobian_det_boundary`	`DIMENSION`	Real	Determinant jacobian at each face for structured grids.
`mass_inverse`	`NNODE`	Real	Inverse of the diagonal mass matrix.
`variables`	`NUM_VARS` $\times$`NNODE`	Real	The updated variables from previous time-step.
`contributions`	`NUM_VARS` $\times$`NNODE`	Real	The computed volume and flux contributions for current time-step.
`auxiliary`	`NUM_AUX` $\times$`NUM_VARS` $\times$`NNODE`	Real	Temporary variable for time integrator.
`materials`	`NUM_MATERIALS` $\times$`NNODE_MATERIALS`	Real	Material properties for each node within an element.
`boundary_conditions`	2$\times$`DIMENSION`	Enum.	Indicates boundary condition: periodic, reflecting, or free surface.

Table B.1: The contents of `ElementDataBase` struct, representing each element in the mesh. Except for `jacobian_det_domain` and `is_element_curvilinear`, other members are arrays of real numbers containing items whose count depends on simulation compile-time configurations.

The `mass_inverse` stores the inverse of the diagonal mass matrix to perform the time integration operation, as shown in Equation (A.32). The `variables` stores the unknown values that are evaluated in the simulation, four for acoustic wave simulation and nine for elastic wave simulation in 3D space, as described in Section 2.2. The `contributions` stores each variable's volume and flux contributions, which will be used to update the `variables` through time integration before advancing to the next time step, as shown in Equation (A.42). Finally, the `materials` stores the material properties for each node within the element. There are two and three material properties for acoustic and elastic wave simulation, respectively. In

this dissertation, each node within an element has uniform material properties (i.e., `NNODE_MATERIALS=1`).

B.2 Simulation Output

The wave simulation application produces several outputs during the runtime. The outputs can help debug the simulation, ensuring it runs as expected. In addition to the output to the terminal (i.e., `stdout`), it can also produce VTK files for visualization purposes using ParaView.

B.2.1 Runtime Output

The runtime output shows the simulation initialization and the progress of the simulation. The output is displayed on `stdout` (i.e., terminal) where the application is run. Figures B.1 and B.2 show the runtime output for acoustic and elastic wave simulations, respectively.

```
[libsc] This is libsc 2.2
[p4est] This is p4est 2.2
[p4est] CPP                      /opt/intel/compilers_and_libraries_2020.4.304/linux/…
[p4est] CPPFLAGS
[p4est] CC                       /opt/intel/compilers_and_libraries_2020.4.304/linux/…
[p4est] CFLAGS                   -g -O2
[p4est] LDFLAGS
[p4est] LIBS                     -lgomp /work/06156/bagus/ls6/program/lapack-3.10.0/…
[PDEBlaster] COMPILER_NAME = GNU
[PDEBlaster] COMPILER_VERSION = 9.4.0
[PDEBlaster] MPI_NAME = /opt/intel/compilers_and_libraries_2020.4.304/linux/mpi/intel…
[PDEBlaster] CUSTOM PREPROCESSOR = PROBLEM_ACOUSTIC PDEBlaster3D PRECISION_SINGLE …
[PDEBlaster] BUILD_TYPE =  Debug
[p4est] Flux solver = 1 (Central: 0, Riemann:1, Penalty continuous:2, Penalty discont…
[p4est] A C O U S T I C   S I M U L A T I O N
[p4est] Into p8est_new with min quadrants 0 level 5 uniform 1
[p4est]  New p8est with 1 trees on 128 processors
[p4est] Done p8est_new with 32768 total quadrants
[p4est] Into p8est_ghost_new FACE
[p4est] Done p8est_ghost_new
[p4est] Step   5 of 1000 (time = 0.000005)
[p4est] Step  10 of 1000 (time = 0.000010)
[p4est] Step  15 of 1000 (time = 0.000015)
[p4est] Step  20 of 1000 (time = 0.000020)
```

Figure B.1: The `stdout` (terminal) output of the application during initialization and runtime steps of acoustic wave simulation. It includes the compiler information, compile-time configurations, runtime configurations, and time-step logging.

```
[libsc] This is libsc 2.2
[p4est] This is p4est 2.2
[p4est] CPP                    /opt/intel/compilers_and_libraries_2020.4.304/linux/…
[p4est] CPPFLAGS
[p4est] CC                     /opt/intel/compilers_and_libraries_2020.4.304/linux/…
[p4est] CFLAGS                 -g -O2
[p4est] LDFLAGS
[p4est] LIBS                   -lgomp /work/06156/bagus/ls6/program/lapack-3.10.0/…
[PDEBlaster] COMPILER_NAME = GNU
[PDEBlaster] COMPILER_VERSION = 9.4.0
[PDEBlaster] MPI_NAME = /opt/intel/compilers_and_libraries_2020.4.304/linux/mpi/intel…
[PDEBlaster] CUSTOM_PREPROCESSOR = PROBLEM_ELASTIC PDEBlaster3D PRECISION_SINGLE …
[PDEBlaster] BUILD_TYPE =  Debug
[p4est] Flux solver = 1 (Central: 0, Riemann:1, Penalty continuous:2, Penalty discont…
[p4est] E L A S T I C   S I M U L A T I O N
[p4est] Into p8est_new with min quadrants 0 level 5 uniform 1
[p4est]  New p8est with 1 trees on 128 processors
[p4est] Done p8est_new with 32768 total quadrants
[p4est] Into p8est_ghost_new FACE
[p4est] Done p8est_ghost_new
[p4est] Did not find a  *.txt file  containing receiver locations in examples/pdeblaster/
[p4est] Step   5 of 1000 (time = 0.000005)
[p4est] Step  10 of 1000 (time = 0.000010)
[p4est] Step  15 of 1000 (time = 0.000015)
[p4est] Step  20 of 1000 (time = 0.000020)
```

Figure B.2: The `stdout` (terminal) output of the application during initialization and runtime steps of elastic wave simulation. It includes the compiler information, compile-time configurations, runtime configurations, and time-step logging.

The first line is a check for `libsc` library used by `p4est`, followed by seven lines that display the check for `p4est` and its compilation configurations. Next, the compile-time configurations of the simulation application are displayed (Appendix B.5), which include the compiler version, the MPI library being used, and the items from compile-time configurations. After that, the runtime configurations are displayed (Appendix B.6), which shows the flux solver being used, the problem type, the refinement level, the number of trees, the number of MPI processes, and the number of elements. Finally, it outputs the time-step log to indicate the simulation status by printing the step number and the time point of the simulation. The frequency of outputs can be controlled through the runtime configurations.

B.2.2 Diagnostic and Statistic Output

At the end of the simulation, the application displays the statistics and diagnostics of the simulation. This helps debug the simulation in case there are numerical or configuration issues. Figures B.3 and B.4 show the diagnostic and statistic output

of acoustic and elastic wave simulations, respectively.

First, it outputs the range of all variables at the end of the simulation: four variables for acoustic wave simulation and nine for elastic wave simulation. It will be easier to spot whether the simulation encounters numerical issues; when the simulation has blown up, the range of the values will not make any sense since it will reach infinity (i.e., very small or very large values). Secondly, there is also numerical error computation, expressed as L_2 error, which compares the computed result against the exact (analytical) solution.

```
[p4est] -----------------------------------------
[p4est] Range of computed solution:
[p4est] --------------------------
[p4est]    p in [-9.999861121177673e-01 , 9.999861121177673e-01]
[p4est]  v_x in [-6.282410118728876e-03 , 6.282409653067589e-03]
[p4est]  v_y in [-6.282409187406301e-03 , 6.282410118728876e-03]
[p4est]  v_z in [-6.282407790422440e-03 , 6.282410118728876e-03]
[p4est] -----------------------------------------
[p4est] Numerical error:
[p4est] ---------------
[p4est]  || v - vh || L2:  1.726723e-05
[p4est] ---------------------------------------------
[p4est] Simulation performance (in seconds):
[p4est] ---------------------------------------------
[p4est]  volume & internal flux:    242.07        55 %
[p4est]  external flux:              34.01         8 %
[p4est]  integrate:                  14.59         3 %
[p4est]  ghost exchange:              0.00         0 %
[p4est]  ghost exchange prep:       152.28        34 %
[p4est]  overhead:                    0.07         0 %
[p4est]  total time:                443.02
[p4est]  ghost exchange begin:       69.68
[p4est]  ghost exchange end:         82.60
[p4est] ---------------------------------------------
```

Figure B.3: The `stdout` (terminal) output of the application at the end of the acoustic wave simulation, showing the range of computed solution for all four variables, numerical error, and time-breakdown by kernel indicating the simulation performance.

Finally, it outputs the simulation performance statistics, which give information on the time spent in each simulation kernel and ghost exchange. This is useful for comparing the performance of CPU and GPU codes and guiding kernel optimizations. The ghost exchange represents synchronous ghost exchange time, while ghost exchange prep represents the asynchronous ghost exchange time. Since it is challenging to measure asynchronous ghost exchange time, at the end of the simulation, the simulation loop consisting of a time-step loop and integration loop (Figure 4.9 is

run without any kernels, only to measure the actual time spent on ghost exchange. Suppose the time measured when the simulation is run for ghost exchange only is larger than the total time measured when the simulation is run with volume kernel (and internal flux, if fused) and ghost exchanges. In that case, there is some overlap between ghost exchange and the kernel execution. Then, by subtracting these values, the actual time spent on ghost exchange (i.e., ghost exchange prep) can be determined, adding to the total simulation time.

```
[p4est] -----------------------------------------
[p4est] Range of computed solution:
[p4est] --------------------------
[p4est]  s11 in [-1.382260990142822e+01 , 1.382261180877686e+01]
[p4est]  s12 in [-1.657705433899537e-05 , 1.411317225574749e-05]
[p4est]  s13 in [-1.797700497263577e-05 , 1.914205677167047e-05]
[p4est]  s22 in [-1.382260513305664e+01 , 1.382260608673096e+01]
[p4est]  s23 in [-8.168140411376953e+00 , 8.168140411376953e+00]
[p4est]  s33 in [-3.015873336791992e+01 , 3.015873336791992e+01]
[p4est]  v_x in [-1.490965041739400e-05 , 1.425133632437792e-05]
[p4est]  v_y in [-6.539769649505615e+00 , 6.539769649505615e+00]
[p4est]  v_z in [-1.256571769714355e+01 , 1.256571674346924e+01]
[p4est] -----------------------------------------
[p4est] Numerical error:
[p4est] ---------------
[p4est]  || v - vh || L2:  6.265332e-03
[p4est] ---------------------------------------------
[p4est] Simulation performance (in seconds):
[p4est] ---------------------------------------------
[p4est]  volume & internal flux:    595.42         62 %
[p4est]  external flux:              94.87         10 %
[p4est]  integrate:                  40.27          4 %
[p4est]  ghost exchange:              0.00          0 %
[p4est]  ghost exchange prep:       233.94         24 %
[p4est]  overhead:                    0.05          0 %
[p4est]  total time:                964.55
[p4est]  ghost exchange begin:      104.62
[p4est]  ghost exchange end:        129.32
[p4est] ---------------------------------------------
```

Figure B.4: The `stdout` (terminal) output of the application at the end of the elastic wave simulation, showing the range of computed solution for all nine variables, numerical error, and time-breakdown by kernel indicating the simulation performance.

B.2.3 Visualization Output

The wave simulation application can output files into the storage disk in a standardized format for visualization. Controlled through the option in runtime configuration, the simulation application can output a VTK file for every several time steps. Using ParaView by Ahrens et al. (2005); Ayachit et al. (2015), this VTK file

can be displayed to visualize and animate the variable change in the problem domain, as shown in Figure B.5. In addition, the application can output the variable values in the form of a text file called a receiver file. This allows plotting the value using MatLab or Spread Sheet program. While tedious, it will enable quick debugging of the simulation result as the time step progresses.

The cube denotes the problem domain space where the simulation is performed to represent the real-world model. The problem domain can be a vast ocean with 100 square kilometers of area and 10 kilometers deep. This has been briefly illustrated in Figure 4.1 where the problem domain is discretized into several elements as part of mesh generation and partitioning (Section 4.1).

B.3 Main Program Execution Flow

This section extends the explanation of Section 4.3.1 by giving more detailed information on the flow of the wave simulation application. Figure B.6 illustrate the high-level flow of the application. The application starts by initializing the MPI library, which includes the initialization of the MPI communicator. Then, `libsc` and `p4est` are initialized by passing the MPI communicator. After these three initialization steps finishes successfully, initialization message is displayed on the `stdout`, as shown in Figures B.1 and B.2.

Next, the context of the problem that will be simulated is initialized. The initialization takes `JSON` input file that stores runtime configuration (Appendix B.6). Then, the simulation engine, where the application will spend most of the time, is initialized. The initialization of the simulation engine is shown in Figure B.7 and will be explained later in this section. After initialization finishes, the simulation engine is run. When the simulation is completed, the simulation engine finalization is performed, which includes cleaning up all memory allocation and object handles. This is followed by the application clean-up, which removes all memory allocations and object handles of `p4est`, `libsc`, and the MPI library.

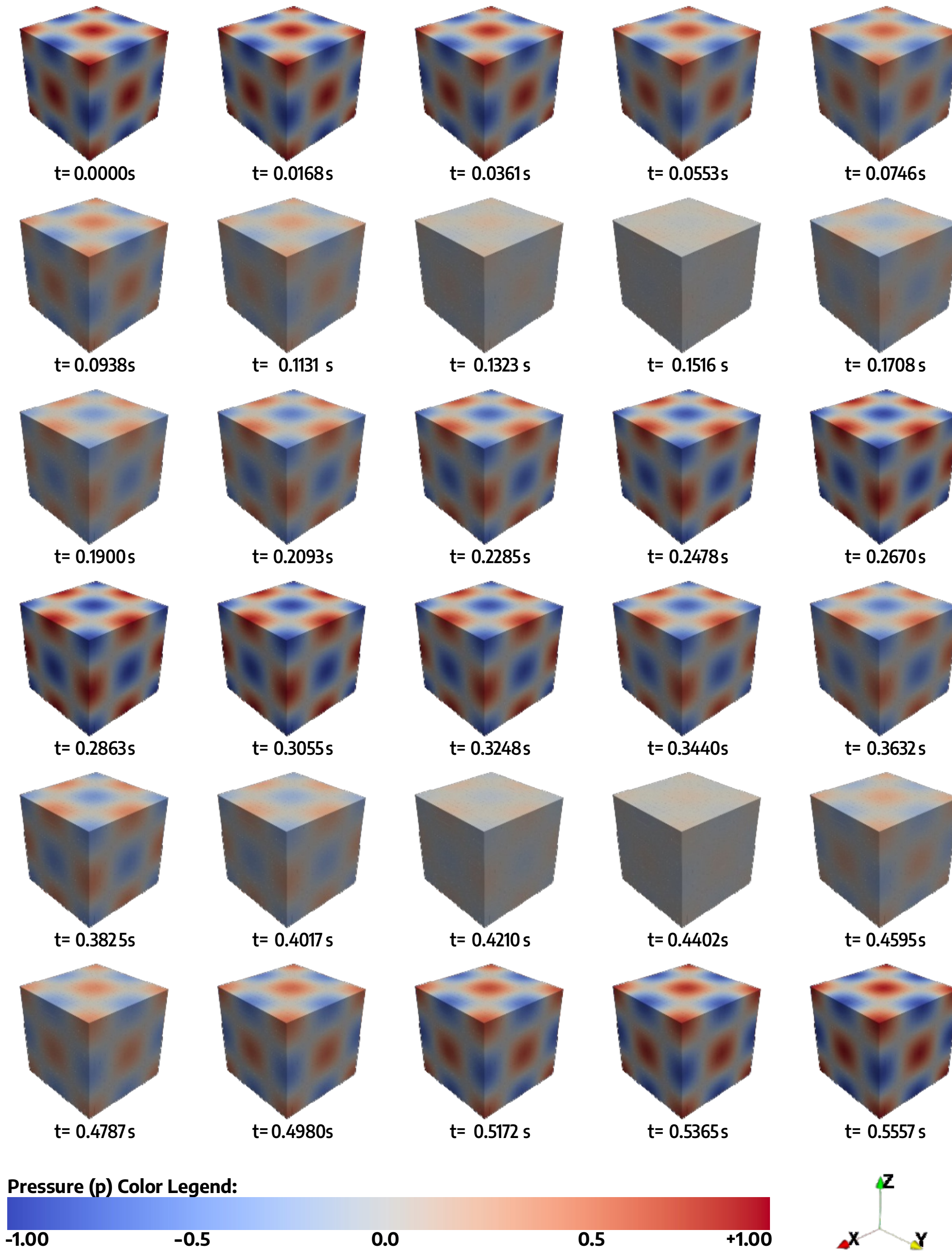


Figure B.5: The example of the output of the wave simulation application, showing the changes in variable pressure, p, as the time step progresses.

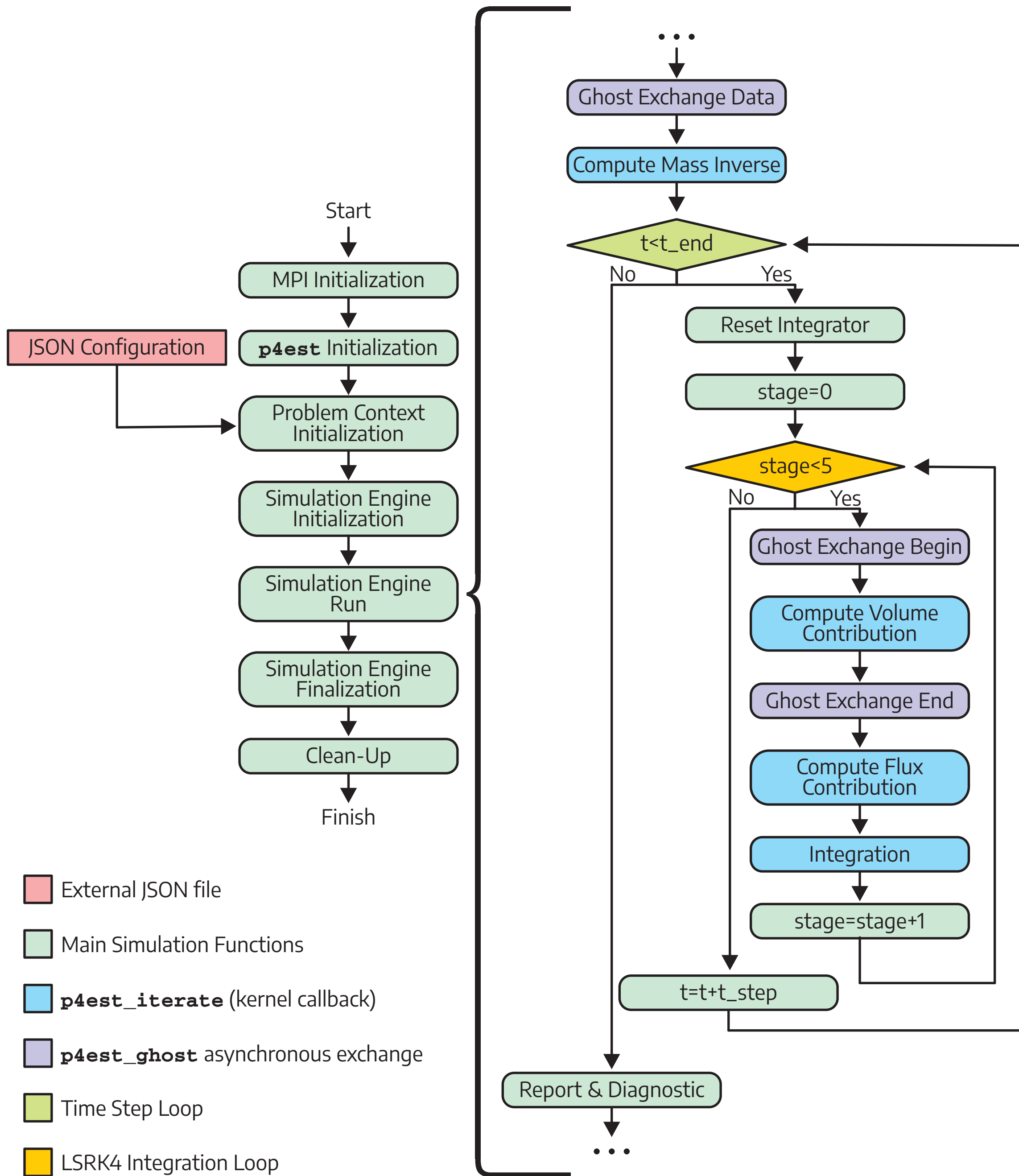


Figure B.6: The high-level flow of the wave simulation application, showing the initialization of external libraries, the configuration of the problem for simulation, initialization of simulation engine, and ending the application.

Inside the simulation engine is where the simulation loop is performed. The right side of Figure B.6 resembles the simulation flow illustrated in Figure 4.9. It starts with performing ghost exchange and running mass inverse kernel before entering the outer loop, the time-step loop. Inside the time-step loop, there is an integration loop consisting of five iterations for the integrator used in this dissertation, the fourth-order Runge-Kutta (LSRK4) integrator (Section 2.3.3). This is where the simulation kernels are executed, including volume kernel, flux kernel, and integration kernel, with which the application spends most of its time. At the end of the simulation engine, there are report and diagnostic that outputs the diagnostic and statistic of the simulation (Appendix B.2.2).

Finally, Figure B.7 shows the lengthy flow of initializing the simulation engine. It starts with `p4est` connectivity initialization. This initialization depends on the type of problem being simulated (i.e., acoustic or elastic wave simulation) and the analytical problem type (i.e., plane wave, lamb wave, Rayleigh wave, or Stoneley wave; Appendix B.6). Then, kernel initialization follows, depending on the problem being simulated. Next, the integrator is initialized; only the LSRK4 integrator is available. After that, the mesh is generated by calling `p4est_new_ext` method and passing the initialization method as a callback.

The initialization method is executed for each quadrant/octant that the MPI process has, setting appropriate values according to the runtime configuration. It is also used to determine the physical space coordinate of each node of the quadrant/octants (`xt`), which will be used to construct the Gauss-Legendre-Lobatto (GLL) integration point for each node. If the element is rectangular, jacobian values need to be computed, which include `jacobian_det_domain`, `jacobian_inverse_domain`, and `jacobian_det_boundary`. Finally, it initializes material properties, state variables, and boundary conditions based on the simulated problem type. During the state variable initialization, the initial value for the `variables` is determined based on the space coordinate (i.e., using `xt`), and both `contributions` and `auxiliary` are set to zero.

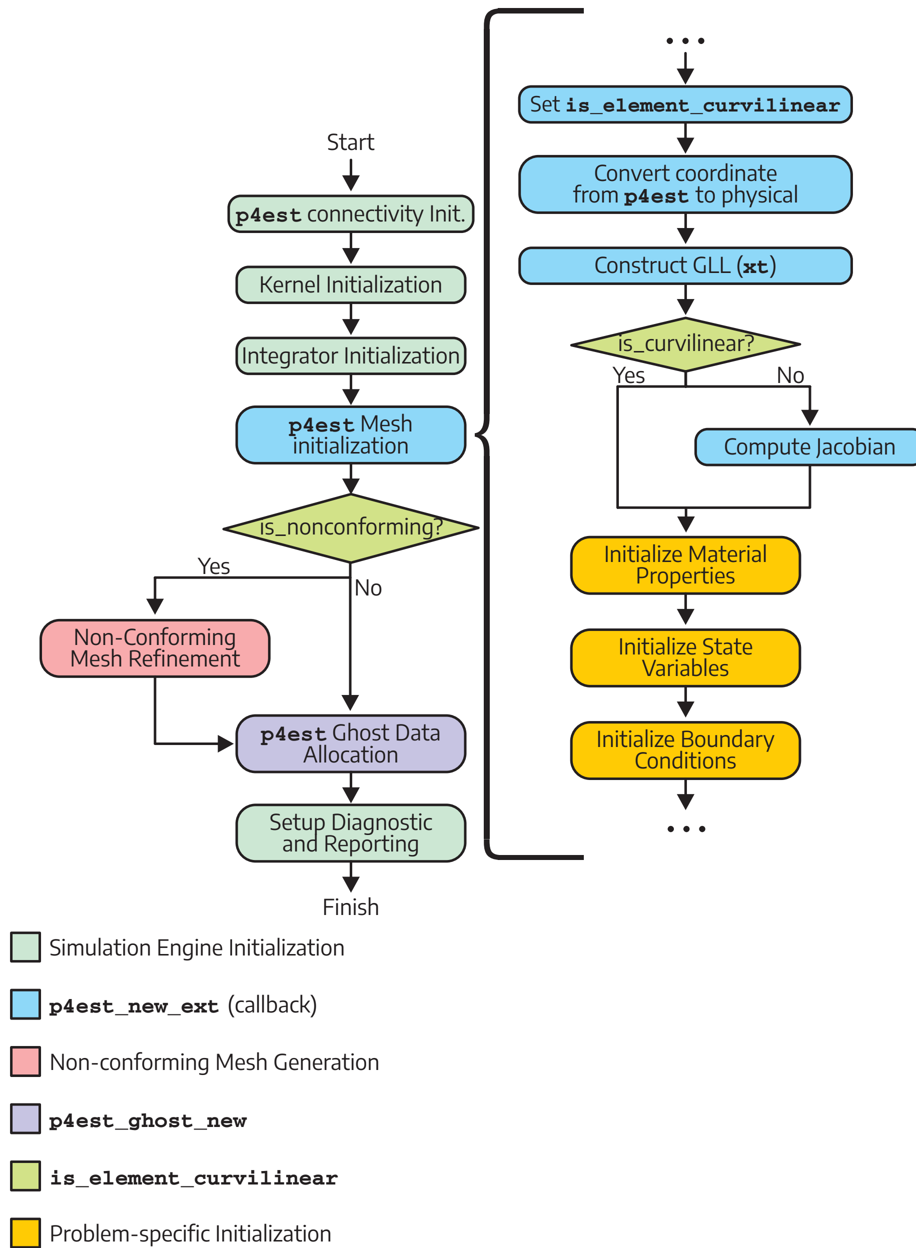


Figure B.7: The high-level flow of the simulation engine initialization, showing the initialization of mesh, elements of mesh, kernel, integrator, ghost exchange, and diagnostic/reporting.

After mesh generation and initialization, further mesh refinement is needed for non-conforming mesh. This includes calling `p4est` methods: `p4est_refine`, `p4est_balance`, and `p4est_partition`. It also computes the inverse of the mass matrix for faces and constructs the filter matrix. However, this dissertation does not consider non-conforming mesh; only uniform mesh is considered. Next, the data structure for ghost exchanges is initialized. This includes allocating memory for the ghost buffer and ghost layer. Finally, the diagnostic and reporting are configured, which will output the information useful for debugging the simulation (Appendix B.2.2).

B.4 Kernel Execution Flow

This section extend the explanation in Section 4.4 by giving more detailed information on the flow of the simulation kernels, which include the mass inverse kernel, the volume kernel, the flux kernel, and the integration kernel.

B.4.1 Mass Inverse Kernel

The mass inverse kernel is used to compute the inverse of the diagonal mass matrix, whose result is stored inside the `mass_inverse` on each element. These values will be used multiple times by the integration kernel. Thanks to the GLL integration scheme, computing the inverse of the mass matrix is a very cheap operation, as shown in Equation (A.35). The kernel is launched once, at the beginning of the simulation, before entering the simulation loop (Figure B.6). Both acoustic and elastic wave simulations have the same mass inverse kernel.

Since this operation is local to each element, the mass inverse kernel is given as `p4est_iter_volume_t` callback to the `p4est_iterate`, allowing the computation to be performed to each element. Figure B.8 shows the high-level flow of mass inverse kernel, consisting of three nested loops corresponding to the problem dimension (i.e., 3D). These three loops are used to iterate over all nodes within an element. The first step is to compute `jacobian_det_w_star`, which needs the `jacobian_det_domain` and

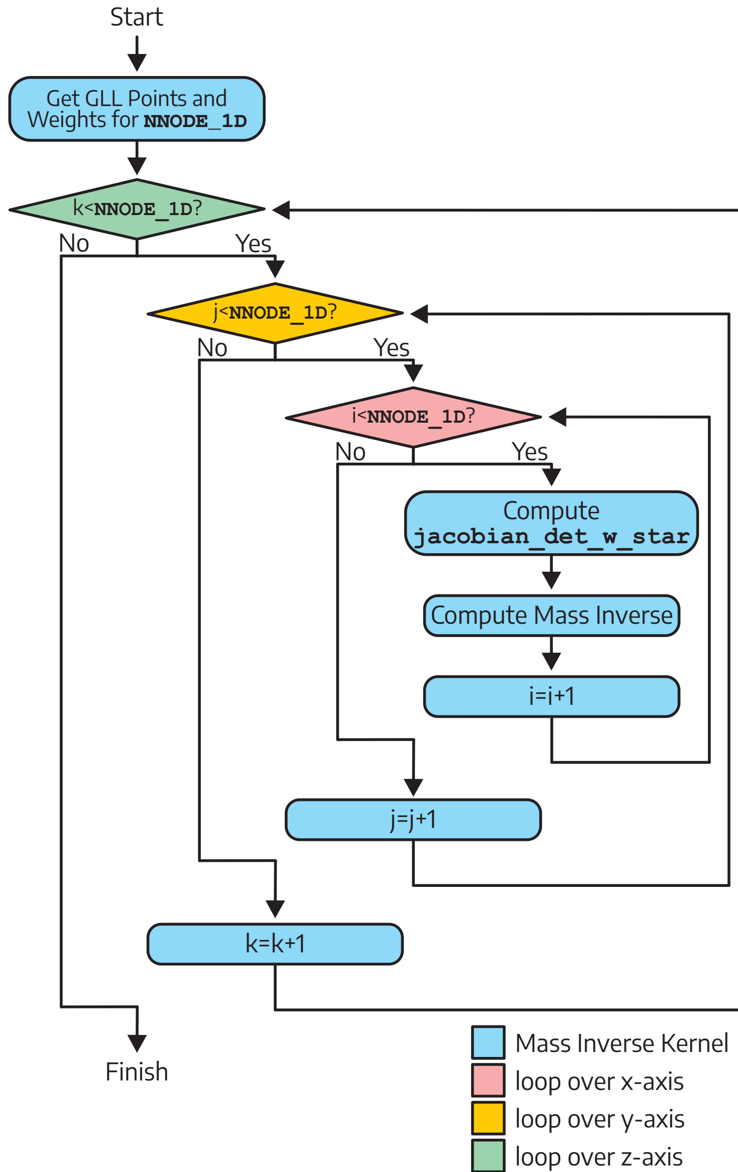


Figure B.8: The high-level flow of the mass inverse kernel, showing the three nested loops that iterate through each node within the element to compute the `mass_inverse`.

GLL weights for a particular coordinate in space. Then, the mass inverse is obtained by calculating the inverse of `jacobian_det_w_star`. Since the kernel is very short and only runs once, it is not considered for optimization.

B.4.2 Volume Kernel

The volume kernel is used to compute the volume contributions and is the most compute-intensive kernel in the simulation. It is given as a `p4est_iter_volume_t` callback to the `p4est_iterate` since it is an entirely local operation. Acoustic and elastic wave simulations have different volume kernels, but the execution structure is similar as shown in Figure B.9. It has three nested loops to iterate through all nodes within the element. At the beginning, the `jacobian_det_w_star` is computed, which needs the `jacobian_det_domain` and the GLL weights. Then, the problem-specific function is called, followed by computing the volume contributions, which are also specific to each problem type. Although the computation is different, the problem-specific function for acoustic and elastic wave simulations have similar flow, shown on the right side of Figure B.9.

B.4.2.1 Volume Kernel for Acoustic Wave Simulation

For acoustic wave simulation, the problem-specific function computes the gradient of pressure (p) and the divergent of particle velocity ($\mathbf{v}$). The name of the function is `compute_div_velocity_and_grad_pressure`. As shown on the right side of Figure B.9, there are three independent loops for every axis of the 3D space. In each loop, the derivative of the shape function is calculated for the axis.

For the x-axis, the **Operation_3** accumulates the result of multiplication between variable $\mathbf{v}_x$ and the derivative of the shape function, resulting in divergent of $\mathbf{v}$. Likewise, the **Operation_4** accumulates the multiplication result between variable p and the derivative shape function, resulting in a gradient of p on the x-axis. The same operations are applied to the y-axis (**Operation_5** with $\mathbf{v}_y$ and **Operation_6**)

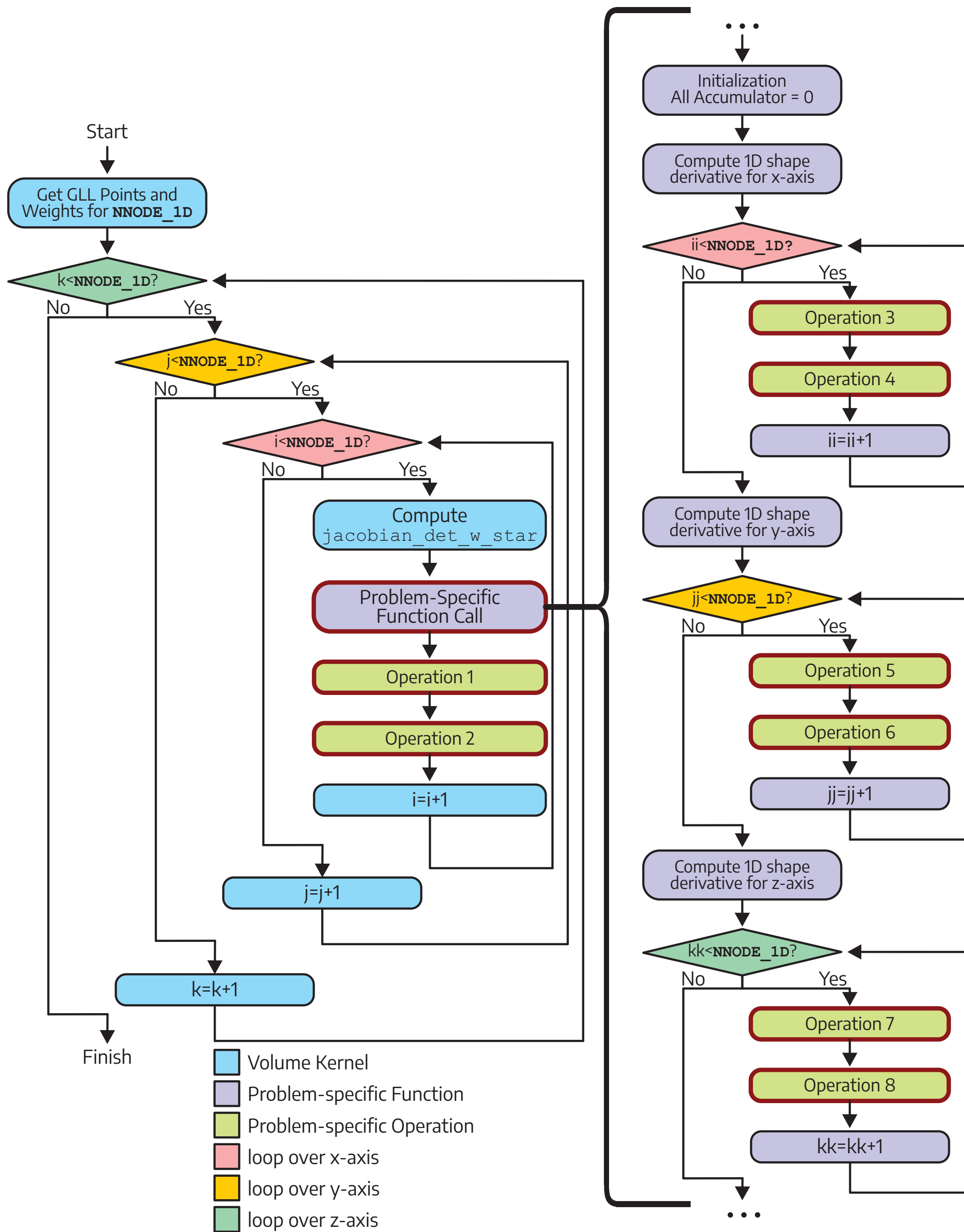


Figure B.9: The high-level flow of the generic volume kernel, showing the three nested loops that iterate through each node within the element to compute the `contributions` of volume. Although Acoustic and Elastic wave simulations have different volume kernels, they share a similar flow. Refer to the text for more details on problem-specific function calls and problem-specific operations.

and z-axis (**Operation_7** with $\mathbf{v}_z$ and **Operation_8**).

After finishing up this function, in **Operation_1**, the divergent of $\mathbf{v}$ is used to compute contributions of p by multiplying it with `jacobian_det_w_star` and the material property κ. Then, in **Operation_2**, the contribution of $\mathbf{v}_x$, $\mathbf{v}_y$, and $\mathbf{v}_z$ are computed by multiplying the inverse of material property ρ (i.e., $1/\rho$), the `jacobian_det_w_star`, and the gradient of p for x-axis, y-axis, and z-axis, respectively.

B.4.2.2 Volume Kernel for Elastic Wave Simulation

For elastic wave simulation, the problem-specific function computes the divergent of stress ($\mathcal{S}$) and the derivative of velocity ($\mathbf{v}$). The name of the function is `compute_div_stress_and_d_velocity`. Like the acoustic version, as shown on the right side of Figure B.9, there are three independent loops for every axis of the 3D space. In each loop, the derivative of the shape function is calculated for the axis. However, the elastic version involves significantly more computation than the acoustic version.

For the x-axis, the **Operation_3** has three accumulation operations: 1) accumulate the result of multiplication between variable $\mathbf{v}_x$ and the derivative of the shape function resulting in the derivative of $\mathbf{v}_{11}$; 2) accumulate the result of multiplication between variable $\mathbf{v}_y$ and the derivative of the shape function resulting in the derivative of $\mathbf{v}_{21}$; and 3) accumulate the result of multiplication between variable $\mathbf{v}_z$ and the derivative of the shape function resulting in the derivative of $\mathbf{v}_{31}$. Likewise, the **Operation_4** has three accumulation: 1) accumulates the multiplication result between variable $\mathcal{S}_{11}$ and the derivative shape function resulting in a divergent of $\mathcal{S}_x$; 1) accumulates the multiplication result between variable $\mathcal{S}_{12}$ and the derivative shape function resulting in a divergent of $\mathcal{S}_y$; and 3) accumulates the multiplication result between variable $\mathcal{S}_{13}$ and the derivative shape function resulting in a divergent of $\mathcal{S}_z$. The same operations are applied to the y-axis (**Operation_5** for $\mathbf{v}_{12}$,$\mathbf{v}_{22}$,$\mathbf{v}_{32}$

and **Operation_6** with $\mathcal{S}_{12}$,$\mathcal{S}_{22}$,$\mathcal{S}_{23}$) and z-axis (**Operation_7** for $\mathbf{v}_{13}$,$\mathbf{v}_{23}$,$\mathbf{v}_{33}$ and **Operation_8** with $\mathcal{S}_{13}$,$\mathcal{S}_{23}$,$\mathcal{S}_{33}$).

After finishing up this function, in **Operation_1**, the divergent of $\mathcal{S}_x$,$\mathcal{S}_y$,$\mathcal{S}_z$ are used to compute contributions of $\mathbf{v}_x$,$\mathbf{v}_y$,$\mathbf{v}_z$, respectively, by multiplying it with `jacobian_det_w_star` and the inverse of material property ρ. Then, in **Operation_2**, the derivative of $\mathbf{v}_{11}$,$\mathbf{v}_{21}$,$\mathbf{v}_{31}$,$\mathbf{v}_{12}$,$\mathbf{v}_{22}$,$\mathbf{v}_{32}$,$\mathbf{v}_{13}$,$\mathbf{v}_{23}$,$\mathbf{v}_{33}$ are used to compute the contribution of $\mathcal{S}_{11}$,$\mathcal{S}_{12}$,$\mathcal{S}_{13}$,$\mathcal{S}_{22}$,$\mathcal{S}_{23}$,$\mathcal{S}_{33}$, which use the material properties λ and μ, and the `jacobian_det_w_star`.

B.4.3 Flux Kernel

The flux kernel is used to compute the flux contributions and is the kernel with a more complex flow than other kernels. It operates at every face of two elements and is given as `p4est_iter_face_t` callback to the `p4est_iterate`. In 3D space, an element may have up to 6 neighboring elements, corresponding to the six faces that an element has. The elements located at the boundary of the problem domain may have fewer neighbors. If neighboring elements are located on a different processor, the data needs to be fetched from the ghost layer instead of local mesh data; this is why ghost exchange needs to be performed before the flux kernel is called. The `p4est_iterate` accepts the ghost layer, which is allocated during the initialization of the simulation engine (Figure B.7) as one of its arguments.

There are several types of flux solvers, such as central, Riemann, penalty, and Lax-Friedrichs. This is given as runtime configuration to the simulation application (Appendix B.6). Furthermore, each problem type also has a different implementation of flux solver. In this dissertation, only the Riemann flux solver is available for Acoustic wave simulation, while the Elastic wave simulation has Riemann and Central flux solvers.

Figure B.10 shows the generic high-level flow of the flux kernels, which is more complicated than other kernels used in this simulation. Flux kernel operates on

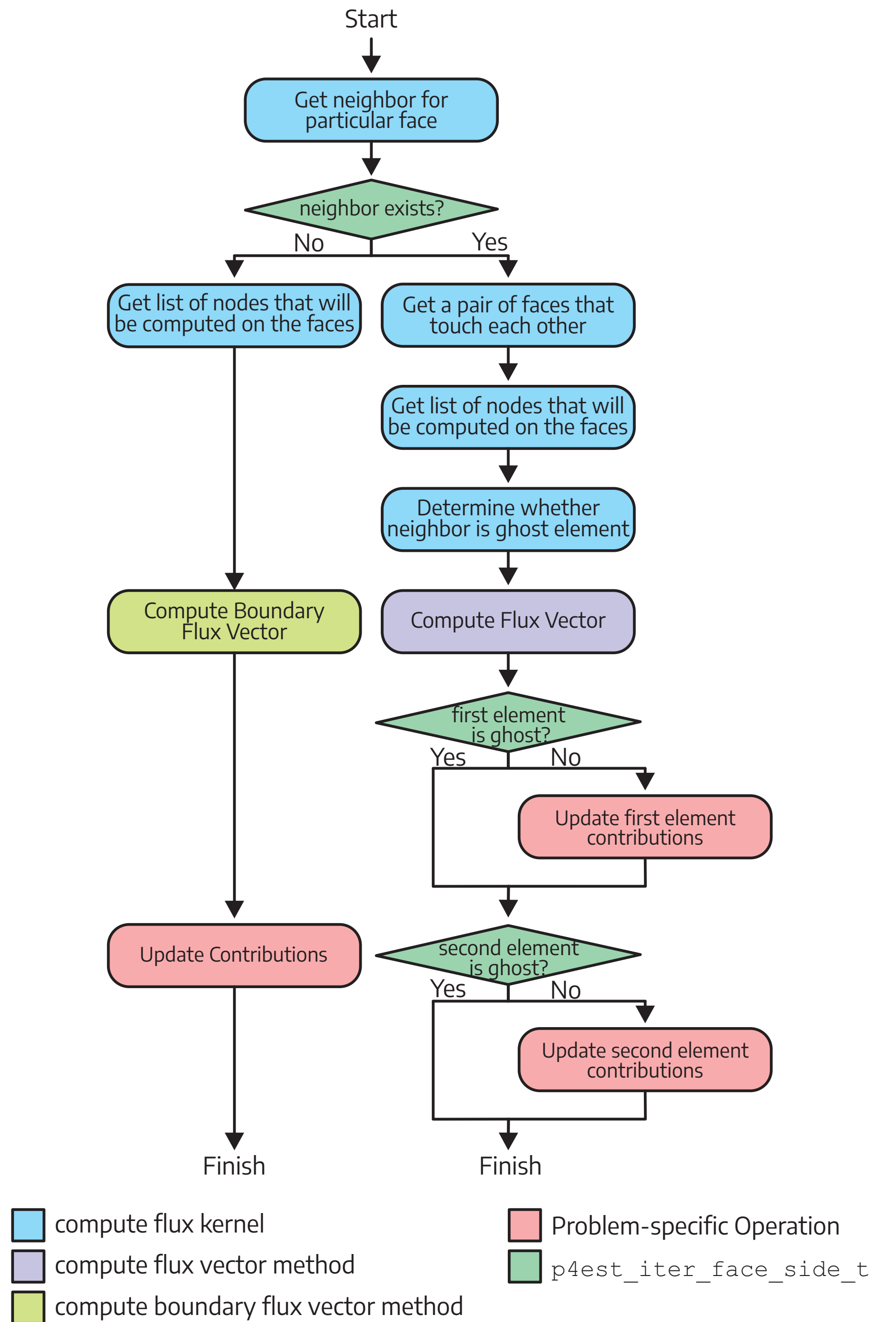


Figure B.10: The high-level flow of the generic flux kernel, showing the two execution paths depending on whether an element has a neighbor or not on a particular face. If the element has a neighbor on its particular face, the right path is taken, where the compute flux vector function is called. Otherwise, the left path is taken, where the compute boundary flux vector function is called.

the nodes located on the face of the element, and it needs a pair of elements that are neighboring each other at that particular face. The `p4est_iterate` will automatically iterate through the pair of faces on which the callback is executed. First, the kernel determines whether a neighboring element exists and accesses its data if it exists. According to the existence of a neighboring element, there are two solver-specific functions: `boundary_flux_vector` to compute the flux contribution for an element located at the boundary with no neighbor on that particular face, and `flux_vector` to compute the flux contribution for an element that has a neighboring element on that specific face.

Before executing the `flux_vector` function, the faces that touch each other from the elements neighboring each other are determined. Then, a list of nodes located on these faces is determined, which will be used to access the `variables` and `contributions` of these nodes. If the neighboring element is a ghost element, then the element's data must be obtained from the ghost layer, which is implemented as an array of `p4est_ghost_t`, instead of the array of `p4est_quadrant_t`. Then, the `flux_vector` function is called to compute the flux contributions. Finally, the `contributions` are updated if only the element is not a ghost element. On the other hand, the `boundary_flux_vector` is simpler since it only handles the face of one element instead of a pair of faces from elements neighboring each other. It does not need to deal with ghost elements and only needs to update the contributions of one element.

B.4.3.1 Flux Vector Function

The flux vector function is a problem-specific and solver-specific function responsible for calculating flux contribution from a pair of faces that touch each other from elements neighboring each other. The right side of Figure B.11 shows the generic flow of the flux vector function.

At the beginning, it obtains the `jacobian_det_boundary` depending on which

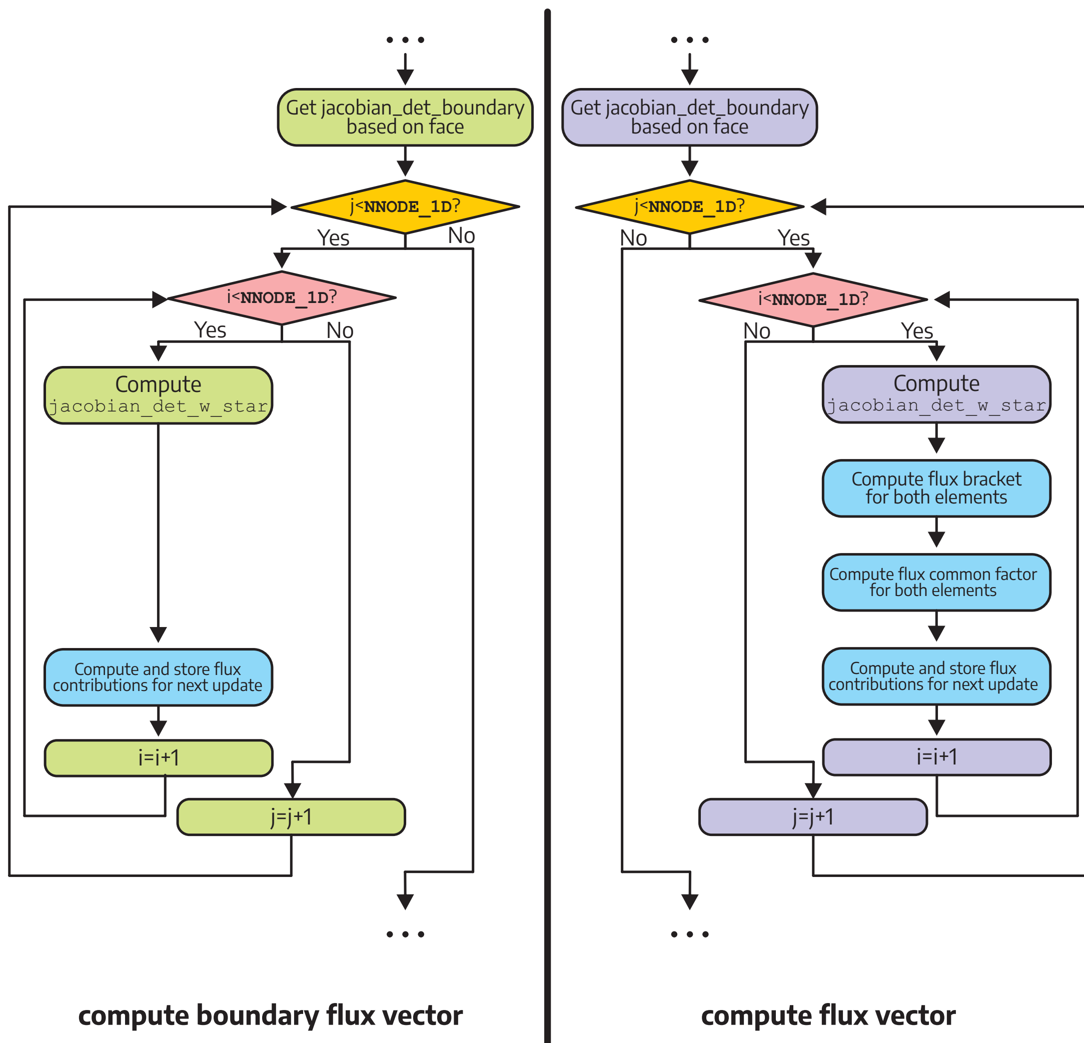


Figure B.11: Depending on whether an element has a neighbor on its particular face, one of the two functions is called to calculate the flux vector: the compute boundary flux vector (left) or the compute flux vector (right). Although both functions are problem-specific, solver-specific, and boundary-condition-specific, they follow the same execution flows.

faces are being computed, according to the face numbering scheme (Figure 4.4). Then, it iterates through all nodes located on the face; hence, only two nested loops are present instead of three. The `jacobian_det_w_star` is still used to compute the flux contributions, which begin by computing the flux bracket for both elements, followed by computing the flux common factor for both elements and finally, computing the flux contributions, which will be accumulated as `contributions` during the update, as shown in Figure B.10. Below is a brief description of the available flux solvers for each problem type used in this dissertation.

- **Riemann flux solver on Acoustic Wave Simulation**.
  - *Bracket:* The p bracket for the first element is computed by subtracting the p of the second element from the p of the first element. The p bracket for the second element can be obtained by multiplying the p bracket with -1. On the other hand, the v bracket is obtained by accumulating each $\mathbf{v}$ bracket according to the axis (i.e., $\mathbf{v}_x$, $\mathbf{v}_y$, $\mathbf{v}_z$). For example, on the x-axis, subtract the $\mathbf{v}_x$ of the second element from $\mathbf{v}_x$, then multiply the subtraction result with the normal vector of that face. The v bracket for both elements is the same.
  - *Common Factor:* First, z_0 and z_1 for the first and second elements, respectively, are computed. The z is the square root of the multiplication of both material properties of the respective element: ρ and κ. The common factor for the first element is obtained by multiplying the inverse of the sum of z_0 and z_1 with `jacobian_det_w_star` and the result of subtracting p bracket of first element with multiplication result of z_1 and $\mathbf{v}$ bracket of the first element. The common factor for the second element is obtained by multiplying the inverse of the sum of z_0 and z_1 with `jacobian_det_w_star` and the result of subtracting p bracket of the second element with multiplication result of z_0 and $\mathbf{v}$ bracket of the second element.

- *Contributions:* Finally, the flux contributions are prepared to update the `contributions`. The flux contributions of p for the first element are obtained by multiplying -1 with the common factor of the first element and material property κ of the first element. The flux contributions of $\mathbf{v}$ are calculated for each $\mathbf{v}_x$, $\mathbf{v}_y$, and $\mathbf{v}_z$ separately by multiplying the common factor of the first element with z_0, the normal vector of the face, and the inverse of material property ρ of the first element. Likewise, the flux contributions of p for the second element are obtained by multiplying -1 with the common factor of the second element and material property κ of the second element. The flux contributions of $\mathbf{v}$ are calculated for each $\mathbf{v}_x$, $\mathbf{v}_y$, and $\mathbf{v}_z$ separately by multiplying the common factor of the second element with z_1, the normal vector of the face, and the inverse of material property ρ of the second element.

- **Riemann flux solver on Elastic Wave Simulation**.
  - *Bracket:* Nine brackets must be computed, corresponding to the number of variables in elastic wave simulation. For example, $\mathcal{S}_{11}$ bracket is obtained by subtracting $\mathcal{S}_{11}$ of the second element from $\mathcal{S}_{11}$ of the first element.
  - *Common Factor:* Many common factors need to be computed. Before computing the common factors, there are several values related to normal vectors, consisting of nz, nz^2, $nx * nz$, $ny * nz$, and $nx * ny * nz$ where nx, ny, and nz are the normal vector of the particular face parallel to the x-axis, y-axis, and z-axis, respectively. In addition, there are several values related to material properties, such as κ, cp, cs, dp, and ds, which represent the multiplication of $\lambda * 2 * \mu$, the square root of κ/ρ, the square root of μ/ρ, the addition of $cp * \kappa$ of the first element and $cp * \kappa$ of the second element, and the addition of $cs * \mu$ of the first element and $cs * \mu$ of the second element. Finally, the common factors can be computed, consisting of 26 that are too complex to describe here.

- *Contributions:* Using the common factors computed earlier, the normal vector values, the material properties values, and the `jacobian_det_w_star`, the flux contributions for all stress $\mathcal{S}$ and particle velocity $\mathbf{v}$ can be computed for each element, consisting of nine variables. The operations are too complex to describe here.

- **Central flux solver on Elastic Wave Simulation**.
  - *Bracket:* Nine brackets must be computed, corresponding to the number of variables in elastic wave simulation. For example, $\mathcal{S}_{11}$ bracket is obtained by subtracting $\mathcal{S}_{11}$ of the second element from $\mathcal{S}_{11}$ of the first element.
  - *Common Factor:* Unlike the Riemann flux solver, the Central flux solver is more straightforward; thus, computing the common factors is unnecessary. Instead, the flux contributions can be computed directly.
  - *Contributions:* Using the brackets, the normal vector values, the material properties values, and the `jacobian_det_w_star`, the flux contributions for all stress $\mathcal{S}$ and particle velocity $\mathbf{v}$ can be computed for each element, consisting of nine variables. The operations are too complex to describe here.

B.4.3.2 Boundary Flux Vector Function

The boundary flux vector function is also a problem-specific and solver-specific function responsible for calculating flux contributions for the face of an element with no neighbors. It also depends on which boundary condition is used, as defined through runtime configuration (Appendix B.6). The left side of Figure B.11 shows the generic flow of the boundary flux vector function.

At the beginning, it obtains the `jacobian_det_boundary` depending on which faces are being computed, according to the face numbering scheme (Figure 4.4). Then, it iterates through all nodes located on the face; hence, only two nested loops are

present instead of three. The `jacobian_det_w_star` is still used to compute the flux contributions, which begin by computing the flux bracket for both elements. However, unlike the flux vector function, the boundary flux vector function is simpler, as it does not need to compute the bracket and common factor for both elements. Below is a brief description of the available flux solvers for each problem type used in this dissertation.

- **Riemann boundary flux solver on Acoustic Wave Simulation**. The Riemann boundary flux vector function for acoustic wave simulation is more straightforward. Since there is only one element, there is no bracket or computation for common factors. Instead, the flux contributions are directly computed. The flux contributions of p are calculated by multiplying -1, material property κ, the normal vector of the face, and `jacobian_det_w_star`. Meanwhile, the flux contributions of $\mathbf{v}_x$,$\mathbf{v}_y$, and $\mathbf{v}_z$ are calculated by multiplying -1, the inverse of material property ρ, z, normal vector of the face, and `jacobian_det_w_star`. Note that z is the square root of the multiplication between κ and ρ.

- **Riemann boundary flux solver on Elastic Wave Simulation**. The Riemann boundary flux vector function for elastic wave simulation is more complex than the acoustic wave simulation. Although only one element is considered, several values related to normal vectors and material properties are still needed, just like the Riemann flux vector function. In addition, several calculations need to be performed based on the boundary condition chosen in the runtime configuration. In this dissertation, only the periodic boundary condition is considered. Then, five common values are computed, which will be used to calculate the flux contributions of all nine variables.

- **Central boundary flux solver on Elastic Wave Simulation**. The Central boundary flux vector function for elastic wave simulation is shorter than the Riemann counterpart. It follows the same flow as the Riemann but with different operations depending on the chosen boundary condition.

B.4.4 Integration Kernel

The integration kernel performs time integration to the `contributions` to update the `variables` for the next time step. In this dissertation, only the fourth-order Low-Storage Runge-Kutta integrator (LSRK4) is available (Section 2.3.3). Like the mass inverse kernel, the integration kernel is short and straightforward, comprising entirely local operations. Thus, it is given as a `p4est_iter_volume_t` callback to the `p4est_iterate`. However, unlike the mass inverse kernel that only runs once at the beginning of the simulation, the integration kernel is run five times per time step. Both acoustic and elastic wave simulations have the same integration kernel.

As explained earlier in Figure B.6, each time step has five integration stages, corresponding to the number of iterations the integration loop has. The high-level flow diagram of the integration kernel is given in Figure B.12. During the 0th stage, the `auxiliary` is initialized to zero, storing the lower-order intermediate results. Then, during each stage, it first compute and accumulate the `auxiliary` based on the previous `auxiliary` values, the `contributions`, the value of time step used, the `mass_inverse`, and the integration coefficient `_rk4a`. Then, the newly updated `auxiliary` is used to update the `variables` by multiplying it with the integration coefficient `_rk4b` and adding it to the previous value of `variables`. The integrator needs the `_rk4c` coefficient if the non-conforming mesh is used.

B.5 Compile-time Configuration

The wave simulation application has several configurations defined during the compile-time by passing the `CMake` variables (i.e., using `-D`). These compile-time configurations determine the compilation properties, simulation properties, GPU acceleration supports, and optimized communication supports. Any changes made to the compile-time configurations require recompiling the simulation program.

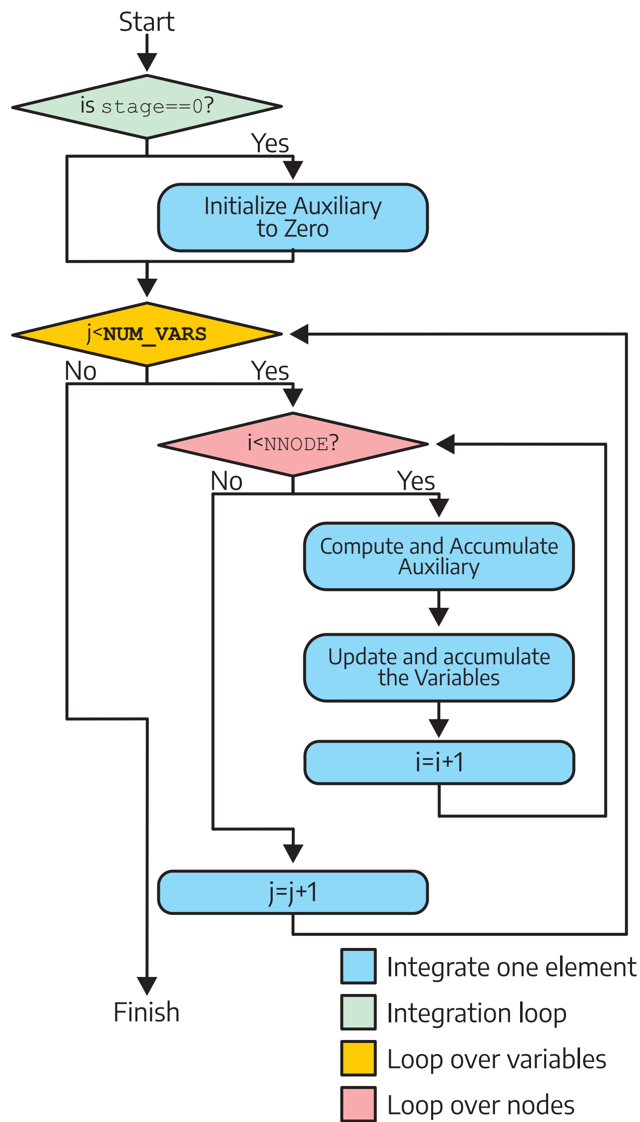


Figure B.12: The high-level flow of the integration kernel, showing the two nested loops that iterate through each variable for each node within the element to update the `variables` based on `contributions`, `auxiliary`, `mass_inverse`, time step value, and integration coefficients. Both acoustic and elastic wave simulations have the same integration kernels, except for the number of variables that need to be integrated per node.

B.5.1 Compilation Dependencies

Before compiling the wave simulation application, it is recommended to check that compatible compilers and libraries are correctly installed on the target system. It requires a compiler that supports `C++ 14` standard (e.g., `gcc` 8.2.0 or higher), `CMake` to generate `makefile` for compilation, with `CMake` version 3.0 for CPU codes and `CMake` version 3.18 for the GPU codes. The following third-party libraries are required to be available and configured properly.

- `p4est`: This wave simulation application uses the adaptive mesh refinement library. As part of `CMake` script, it will automatically compile the `p4est` before building the application. This dissertation uses `p4est` 2.2; a newer version should also work.
- `catch`: This library is used as a test framework for CPU code, providing multiple test paradigms. This dissertation uses `catch` 2.2.3.
- `protoproc`: This library is used as a pre-processor to create a C++ class defining the problem context of the simulation, which is derived from the compile-time configurations (Appendix B.5). This dissertation uses `protoproc` 0.3.
- `lapack`: This is the linear algebra package used by the CPU code of the simulation. This dissertation uses `lapack` 3.9.0.
- MPI library: This is required for both `p4est` and the simulation application. For GPU code, uses MPI library that is CUDA-aware and built with CUDA support.

B.5.2 Compilation Properties

Below are the compile-time configurations that control the behavior when compiling the simulation. It allows the user to choose which compiler toolchain to compile the simulation and the compiler optimization level.

- Compiler toolchain, which can be configured to use either `gcc` or vendor-supplied toolchain, such as Intel Compiler (`icc`). This is controlled through `CMake` variables `-DCMAKE_C_COMPILER` and `-DCMAKE_CXX_COMPILER`.

- The build mode, which can be either `Debug` or `Release`. For debugging purposes, it is recommended to choose `Debug`, which enables compiler features that make it easy to debug. However, choosing `Release` will yield better performance for production runs since the compiler will aggressively optimize the code. This is controlled through `CMake` variable `-DBUILD_MODE`.

- Automatic testing mode, which can be either `ON` or `OFF`. If it is enabled, then, after compilation, automatic unit tests are run to ensure the compilation is successful and the host machine configuration is correct to run the simulation. This is controlled through `CMake` variable `-DENABLE_TESTING`.

- The number of CPU threads to run for automatic testing mode, which can be controlled through `CMake` variable `-DTEST_NUM_PROCESSES`.

B.5.3 Simulation Properties

Below are the compile-time configurations that control the properties of the simulations, which include the type of wave equations being solved, the problem dimension, the precision of computation, the number of nodes per element, and the number of material nodes per direction.

- Type of the wave equations (i.e., problem type), either acoustic or elastic wave equations. It is defined through `CMake` variable `-DPROBLEM_TYPE`.

- The dimensionality of the problem (i.e., problem dimension), either two-dimension (2D, use `p4est`) or three-dimension (3D, use `p8est`). It is defined through `CMake` variable `-DDIMENSION`.

- The precision used for computation. The earlier version only supports double precision (FP64) and single precision (FP32) for running the simulation. The newer version added half-precision, either IEEE 754 FP16 precision, as described in Institute of Electrical and Electronics Engineers (2019) standard, or BFloat16 precision, as described by Wang and Kanwar (2019). The precision can be controlled through `CMake` variable `-DPRECISION`. Only FP64 and FP32 are discussed in this dissertation.

- The number of nodes that each element has. It can be controlled through `CMake` variable `-DNNODE_1D`. For example, if the 512-node element is desired for a problem in 3D space, then `-DNNODE_1D=8`, as explained in Section 4.1.2.

- Number of material nodes in each direction, which can be constant, linear, or the same as `NNODE_1D`. It is defined through `-DNNODE_MATERIAL_1D`. In this dissertation, only `-DNNODE_MATERIAL_1D=1` is considered.

B.5.4 GPU-Acceleration Support

Below are the compile-time configurations that control the GPU acceleration support, which is added through the work described in Chapter 5.

- The support for GPU acceleration, which can be controlled through `CMake` variable `-DENABLE_CUDA`. If GPU acceleration is enabled, the GPU codes are automatically compiled using the NVIDIA CUDA Compiler (`nvcc`).

- The GPU kernel selection corresponds to the basic or optimized kernels as explained in Sections 5.3 and 5.4. The GPU kernels can be chosen through `CMake` variable `-DGPU_KERNEL`. The `GPU_base` uses `KERNEL_STDV1`, while `GPU_fl` and `GPU_fls` use `KERNEL_FUSED` and `KERNEL_FUSEDSHARED`, respectively.

B.5.5 Optimized Communication Support

Below are the compile-time configurations that control the GPU ghost exchange optimization, which is added through the work described in Section 5.6.2 and further extension described in Chapter 6.

- The element database used for ghost exchange, either the standard element database `ElementDataBase` or the reduced size `ElementDataBaseGhost`, as explained in Section 5.6.2. The `CMake` variable responsible for making this choice is `-DGPU_GHOST_ELEMENT`, either `GHOST_STD`, `GHOST_OPT1`, or `GHOST_OPT2`. The `GHOST_OPT1` is discussed in Section 5.6.2 while `GHOST_OPT2` is discussed in Section 6.3.
- The choice of precision used for ghost exchange consists of two types: the ghost exchange computation precision and the ghost exchange communication precision, defined through `CMake` variables `GPU_GHOST_CPRECISION` and `GPU_GHOST_EPRECISION`, respectively. The precision can be double (FP64), single (FP32), or half-precision (FP16 or BFloat16). However, only FP64, FP32, and FP16 are used in this dissertation.

B.6 Runtime Configuration

As its name suggests, the runtime configuration controls how the wave simulation progresses. In contrast to compile-time configurations, it does not need to recompile the simulation program. Instead, the configurations are given to the simulation program as an input file in JavaScript Object Notation (JSON) format, which is then used to initialize the problem to be simulated. The configurations defined through the JSON file are briefly explained below.

- `ProblemType`, which indicates whether it is an acoustic or elastic wave simulation. Note that, this must align with the compile-time configuration defined when compiling the program.

- `AnalyticalProblemName`, which indicates the boundary/initial conditions for a fixed analytical solution and compare-solution. For acoustic wave simulation, only plane waves are implemented. For elastic wave simulation, there are several choices: plane wave, lamb wave, Rayleigh wave, and Stoneley wave. This dissertation considers only the plane wave for GPU acceleration.

- `BoundaryConditions`, which defines the boundary conditions, either periodic, reflecting, or free surface. This dissertation considers periodic boundary conditions.

- `DomainDimensions`, which controls the problem domain sizes (i.e., the number of trees), in (x, y, z) configuration. By default, the dimension is $(1, 1, 1)$, which consists of one tree representing one cube (e.g., as shown in Appendix B.2). A problem domain can be represented using multiple trees, such as $(2, 2, 1)$, which uses four trees (four cubes), resulting in a cuboid-shaped problem domain. However, the problem domain with $(2, 2, 2)$ is cube-shaped with eight trees (eight cubes). The number of elements in each cube is equal and is controlled by the refinement level.

- `AreElementsCurvilinear`, which indicates whether the elements are rectangular or curvilinear.

- `ComputeDiagnostics`, which determines whether simulation diagnostic needs to be performed at the end of the simulation. The diagnostics include computing the solution ranges and the L2 error, which help observe any numerical instability issue with the simulation.

- `FluxSolver`, which allows to select different flux solvers when performing flux computation. Only the Riemann flux solver is available for acoustic wave simulation. For the elastic wave simulation, central, Riemann, penalty continuous, penalty discontinuous, and Lax-Friedrichs flux solvers can be chosen. However, only central and Riemann flux solvers are discussed in this dissertation.

- `StartTime`, which determines the start time of the simulation.
- `EndTime`, which determines the end time of the simulation.
- `TimeStep`, which determines the time-step size for simulation.
- `VisualizationFileNamePrefix`, which determines the prefix of the VTK files used for dumping variables for visualization purposes, such as using ParaView as shown in Appendix B.2.
- `VisualizedFields`, which determines the variables that needs to be stored in VTK files for visualization purposes.
- `WriteOutputFrequency`, which controls how often (in time steps) the VTK file is generated. Zero means no VTK file is generated. Non-zero N value means the VTK file is generated for every N time steps. Frequently generating VTK value incurs additional overhead, especially for GPU acceleration, since the mesh data must be copied back to the CPU memory before the VTK can be generated.
- `MinimumQuadrantsPerCore`, which determines the minimum number of elements distributed to each thread. Zero means there is no minimum number of elements.
- `RefinementLevel`, which determines the refinement level of the mesh (Section 4.1.1). This also determines the number of elements in a tree (cube).
- `FillUniform`, which indicates whether the mesh is uniformly filled or not.
- `IsMeshNonConforming`, which indicates whether the mesh is non-conforming or not (Section 2.3.1).
- `GmshFileName`, which defines the input mesh file in Abqus format, generated by an external program, `Gmsh` by Geuzaine and Remacle (2009), instead of generated by `p4est`.

- `ReceiversFileName`, which defines the output receivers file in ASCII format, where it dumps the motion of the point (i.e., velocity) for plotting purposes.

- `Epsilon`, which defines the small number used for comparing floating-point numbers.

- `TimeStepLogFrequency`, which controls how often (in time steps) the simulation log is printed out to the standard output.

Glossary

Acoustic Waves A mechanical wave longitudinally travels through fluid or gas (i.e., the particles' movement is parallel to the propagation direction; also known as compressional waves) where only pressure variations occur. It is modeled using acoustic wave equations with four unknown variables shown in Section 2.2.1.

Adaptive Mesh Refinement Computational method to increase the efficiency and accuracy of numerical simulations by dynamically adjusting the resolution of the computational grid (mesh) based on the complexity of the solution in different regions of the problem domain. See Section 2.4.1.

ALU See **Arithmetic Logical Unit**.

AMR See **Adaptive Mesh Refinement**.

API See **Application-Programming Interface**.

Application-Programming Interface Set of rules, protocols, and tools that allow applications to communicate with each other. In the context of developing applications for specific hardware, it allows the users to call, use, and interface with low-level libraries, drivers, and frameworks provided by the hardware manufacturer.

Arithmetic Intensity Measures the amount of computation (i.e., FLOP/s) that can be done per data byte. See Section 2.5.2.

Arithmetic Logical Unit A fundamental microprocessor component consisting of digital circuits that perform arithmetic and logical (bitwise) operations on binary data.

Asynchronous Progression A feature specific to a particular MPI implementation allows the communication to progress by using a dedicated thread, freeing the

main thread to perform computation. Although it is not guaranteed by standard, and thus, the implementation may vary, it allows overlapping communication with computation when using non-blocking communication functions when enabled. See Section 5.6.3.

`auxiliary` An array with a length of `NNODE`×`NUM_VARS` inside the element data structure that stores the temporary data for LSRK4 temporal integration. See Sections 2.3.3 and 4.2.3 as well as Appendix B.1.

Basic Linear Algebra Subprograms Standardized collection of functions/routines that perform common linear algebra operations, providing low-level building blocks for performing operations in scientific computing, machine learning, and numerical simulations. Level-1 BLAS involves vector-vector operations, while Level-2 and Level-3 BLAS involve vector-matrix and matrix-matrix operations, respectively.

Batching A technique to divide the data into multiple subsets, allowing it to operate on particular subsets at a time due to insufficient memory. The batching technique is used to improve the scalability of the PIM chip, allowing it to handle problems that are larger than the capacity of the PIM chip, as described in Section 7.3.1.

BLAS See **Basic Linear Algebra Subprograms**.

Block See **Thread Block**.

Blocking Communication Communication functions in MPI where the thread must wait until the message transmission achieves a particular state before executing the following instructions. See Section 2.4.2.

Boundary Element Elements that are located at the edge, face, or corner of the problem domain. These elements do not have neighboring elements for one,

two, or three faces and need special handling for computing flux contributions as shown in Figure B.10.

Bulk Modulus Material property, denoted by κ, that represents a substance's resistance to uniform compression. It quantifies the decrease of volume when subjected to an increase in pressure while maintaining equal compression in all directions. See Section 2.2.1.

Bus In the context of interconnection, it is the simplest topology where all endpoints (e.g., components) share the same path for data communication. Thus, only one communication can take place at a time. The central switch acts as an arbiter to control which endpoints can reserve the bus and perform their communication. Bus is one of the proposed interconnects for processing-in-memory, as described in Section 7.1.3.3.

Cache Small size, high-speed, low-latency, hardware-managed on-chip memory placed near the ALU of a processor to store frequently-used data, speeding up overall operations since they do not need to be fetched multiple times from the significantly slower off-chip memory. A processor can have several levels of cache (hierarchical cache), comprising first-level cache, middle-level cache, and last-level cache. The former is the closest to ALU, the fastest to access but the smallest in capacity, while the latter is the largest but slower to access. The applications must have cache-friendly memory access patterns leveraging temporal and spatial locality to take advantage of caches.

Callback Function Function that is given as an argument to another function and is invoked (i.e., called back) later after some events or operations have been completed. In the context of this dissertation, the `p4est_iterate` accept callback functions to operate on the interior, face, and corner of an element, as described in Section 4.2.2. The callback functions can colloquially be called

simulation kernels since they operate on an element's interior (volume kernel) and face (flux kernel).

CC See **CUDA Cores**.

Central Processing Unit . A general-purpose microprocessor that performs the tasks of processing data and executing instructions. It can handle applications with complex execution flow and memory access patterns. It can also have multiple cores for parallel computation. In the context of this dissertation, the central processing unit is a host processor with its memory. The accelerator devices, such as GPUs and PIMs, are attached to it through a communication bus (e.g., PCI Express), allowing the host and device to exchange data. The model of the central processing unit used in this dissertation includes Intel Xeon Platinum 8160 (x86-64), AMD EPYC 7742 (x86-64), IBM POWER9 (ppc64le), and ARM Cortex A72 (AArch64).

CIM See **Compute-in-Memory**.

Cluster In the context of high-performance computing, it refers to a group of computers, which are called compute nodes or simply nodes, that are interconnected through a fast network, allowing them to work together as a large system, aggregating computing power and memory to solve complex, large-scale computational problems. The workloads are distributed across multiple compute nodes, enabling parallel processing and accelerating computational tasks.

Collective Communication An MPI communication function involves all MPI communicator members working and manipulating shared information, including synchronization (e.g., barrier), data movement (e.g., broadcast, gather, scatter, all-gather, all-to-all), and global computation (e.g., reduce, all-reduce, scan). See Section 2.4.2.2.

Column Buffer Temporary storage inside memory block to store recently-read column, allowing fast access to data if the subsequent accesses go into the same column.

Column Driver Circuit to select a specific column within the memory array, activate it, and copy the data into the column buffer, allowing data to be read or written.

Column-Parallel Execution A parallel execution inside a memory block where all columns perform the same operations but with different data, assuming different data is stored in each column. It can be viewed as a single-instruction multiple-data for the memory block. See Section 2.7.2.

Communicator A group containing MPI processes that can communicate with each other. The communicator defines the context in which the communication occurs (e.g., ranks) and determines which processes could participate, including collective communication. The default communicator that contains all MPI processes is called `MPI_COMM_WORLD`. Users can create custom communicators and can control how MPI processes are grouped. One MPI process can belong to multiple communicator groups.

Compute Bound An application whose performance is primarily limited by the hardware's processing power.

Compute Node A standalone computer comprising CPUs, memory, accelerators (e.g., GPUs, FPGAs), and local storage that perform computation. Many compute nodes are connected through a fast interconnect (e.g., InfiniBand) to form a supercomputing cluster. This allows them to provide aggregate computing power and memory for large-scale computational tasks.

Compute Throughput Measures the number of floating-point or integer operations performed per second by the hardware. See Section 2.5.2.

Compute Unified Device Architecture Parallel computing platform and programming model developed by NVIDIA for their general-purpose Graphics Processing Units. It is often called CUDA and allows users to build applications that take advantage of massively parallel execution of GPUs, including scientific simulations, machine learning, and image processing.

Compute-in-Memory An emerging computing paradigm where the computation is performed directly within the memory hardware, where the data is stored, eliminating the need to transfer data back and forth between memory and processor. There are two approaches to compute-in-memory: near-memory-processing (NMP) and processing-in-memory (PIM). See Section 2.7.

Conforming Mesh A type of computational mesh used in numerical simulation where the mesh elements are aligned precisely with the geometry of the problem domain and each other, resulting in a mesh that has no gaps, overlaps, and mismatched edges between adjacent elements.

Constant Memory A read-only memory in GPU to store data that remains constant throughout the execution of GPU kernel. It provides high-speed, low-latency access to frequently used data for all threads in the kernel grid. It is suitable for storing physical constant, look-up tables, and kernel configurations. See Section 2.6.3.1.

`contributions` An array with a length of `NNODE`×`NUM_VARS` inside the element data structure that stores and accumulates volume contributions (i.e., the result of volume kernel) and flux contributions (i.e., the result of flux kernel). See Section 4.2.3 as well as Appendix B.1.

CPU See **Central Processing Unit**.

CTA Cooperative-Thread Arrays; See **Thread Block**.

CUDA See **Compute Unified Device Architecture**.

CUDA Cores The term NVIDIA uses to call the vector ALUs inside their GPUs, although the term *cores* is different than the cores in the CPU. The CUDA Cores are the main workhorse of NVIDIA GPUs; thousands of them allow massive parallel operations. The number of FP32 ALUs inside the GPUs usually represents the number of CUDA Cores. Datacenter-class GPUs, such as NVIDIA Tesla P100, NVIDIA Tesla V100, and NVIDIA A100, are equipped with a significantly large number of FP64 CUDA Cores compared to consumer-class GPUs, allowing them to run high-performance scientific applications that require double precision. See Section 2.6.2.

CUDA-Aware A feature specific to a particular MPI implementation that recognizes whether memory buffers are stored in CPUs or GPUs memory. Using this feature, MPI can directly send and receive data to or from memory buffers stored in GPU without the need to stage them first in CPU memory, significantly reducing the communication overhead between CPU and GPU. See Section 2.4.2.4.

Data Hazard A type of hazard in computing where the next instruction depends on the results of a previous execution that has not yet completed its execution. Data hazard includes Read-After-Write (i.e., when an instruction reads (old) data before a previous instruction writes it), Write-After-Read (i.e., when an instruction writes the data before a previous reads (old) data), and Write-After-Write (i.e., when an instruction writes new data before a previous instruction writes it). The last two occur in concurrent execution.

Decoder A circuit inside a memory chip that interprets the address and instruction sent by the host CPU to select a specific memory block for particular operations.

Density Material property, denoted by ρ, that measures how much mass is contained within a given volume of a material.

Device Memory A memory owned by the device (i.e., GPUs). In the context of this dissertation, it is a separate memory that is only accessible by kernel runs on GPUs. Before running a kernel, necessary data must be copied from the host memory (i.e., CPU memory) to the device memory (i.e., GPU memory). At the end of kernel execution, the result is copied from the device memory to the host memory. NVIDIA provides Unified Virtual Memory (UVM) to automate this process, where the driver handles the data copy between host and device memories.

DG See **Discontinuous Galerkin**.

Direct Memory Access A technique that allows the devices (e.g., network interface cards, graphics processing units) to transfer the data from or to the host memory without the involvement of the CPU, freeing it to work on another task (i.e., computation).

Discontinuous Galerkin A numerical method used to solve partial differential equations (PDEs) that allows the solution to be discontinuous between neighboring elements. It divides the problem domain into discrete elements whose solutions are computed independently (i.e., local to each element). Interactions between neighboring elements are handled by specific numerical fluxes at the interfaces, which take care of the discontinuity. DG has lower inter-element communication costs than the Finite Difference Method (FDM) and Spectral Element Method (SEM). See Section 2.3.1 and Appendix A.3.

Discretization A process of transforming continuous models, such as differential equations and integrals, into discrete forms that can be solved using numerical methods. The continuous models can describe real-world phenomena that change over time and space. Since it is impossible to compute solutions for every point in space, discretization is used to solve the problems using computers by computing the solutions at discrete points in space and time.

Divergence A mathematical operator that measures the rate at which a vector field flows from a point, indicating the presence of sources or sinks in the fields. The operator takes a vector field as input and outputs a scalar value.

DMA See **Direct Memory Access**.

Double Precision In computation, it refers to using a 64-bit floating-point format to store and perform arithmetic operations on real numbers. Double precision provides higher numerical accuracy for scientific computations, simulations, and numerical analysis.

DRAM See **Dynamic Random Access Memory**.

DUAL A work by Imani et al. (2020) that provides the energy model for memristor-based digital PIM. See Section 3.3.3.

Dynamic Random Access Memory A random access memory technology that uses electrical charges inside capacitors to store the data (e.g., binary '1' for the presence of charge while binary '0' for the absence of charge). Since the electrical charges inside the capacitors slowly fade over time, periodic refresh is required to maintain the data integrity. It offers higher bit density than static random access memory (SRAM).

EDR Enhanced Data Rate, which is an InfiniBand standard released in 2014 that provides 100 Gbps bandwidth in each direction.

Elastic Waves A type of wave that propagates through an elastic medium (e.g., solid, liquids, gases) by causing the particles of the medium to oscillate due to the material's ability to deform under stress and return to its original state when the stress is removed (i.e., elasticity). All acoustic waves are considered elastic waves, but not vice versa.

Element In the context of numerical analysis, it is a fundamental building block of a computational mesh; it is a small piece resulting from discretization in which the solution of the partial differential equations is computed. See Section 4.1.2.

Element Node Discrete point in the element's space where the `variables` are explicitly computed. See Section 4.1.2.

Element Order Refers to the degree of the shape functions (i.e., interpolation functions) used to approximate the solution within the element. The order of the element is directly related to the degree (i.e., polynomial order) of the shape functions. See Appendix A.3.3.

Element Processor A terminology used by Gourounas et al. (2023b,a) to call a custom hardware structure designed to process one element. See Section 6.1.1.2.

Empirical Roofline Toolkit A framework by Yang (2015) to create the roofline model of computing hardware through measurement (i.e., empirically). See Section 2.5.2.

ERT See **Empirical Roofline Toolkit**.

Expansion A technique to split an element and store the slice in multiple memory blocks. This is required for elastic wave simulation since single memory block is insufficient to store the element. In addition, for problem sizes smaller than the capacity of the PIM chip, splitting an element into multiple memory blocks increases parallelism since each `variables` can be computed in parallel. See Section 7.3.2.

Face In the context of numerical analysis, it refers to the boundary or surface of an element where it interacts with neighboring elements or the external environments. Faces are critical for defining the connectivity between elements, applying boundary conditions, and solving problems involving flux or surface integrals.

Fat-tree A network architecture commonly used in HPC to connect the compute nodes in a cluster using a hierarchical tree-like structure where links near the root have higher bandwidth than those near the leaves (i.e., the compute nodes). It is called a fat tree because the links nearer to the top are fatter (e.g., thicker, having more bandwidth) than links further down the hierarchy.

FDM See **Finite Difference Method**.

FDR Fourteen Data Rate, which is an InfiniBand standard released in 2011 that provides 54 Gbps bandwidth in each direction.

Field-Programmable Gate Array A type of chip that, unlike traditional chips where the logic is fixed, can be configured by the user after manufacturing. This allows users to develop custom logic circuits tailored for specific applications.

Finite Difference Method A numerical method used to solve differential equations by approximating derivatives with finite differences (e.g., $f(x+b) - f(x+a)$). The spatial (i.e., space) and temporal (i.e., time) domains are discretized into a finite number of intervals. The solution at a particular point of the intervals is approximated by solving algebraic equations containing finite differences and values from nearby points. It is used in engineering, physics, and applied mathematics, such as heat transfer, fluid dynamics, and structural analysis.

Floating-Point Number A data type used in the computer to represent real numbers with decimal places, where the decimal point can *float* to different positions.

Floating-Point Operation Operation involves calculating floating-point numbers.

Floating-Point Operation per Second The number of floating-point operations that can be performed per second. It is commonly used to represent the performance of computer hardware (e.g., CPUs, GPUs).

FloatPIM A cycle-accurate simulator of memristor-based Processing-in-Memory system developed by Imani et al. (2019a). See Section 3.3.1.

FLOP See **Floating-Point Operation**.

FLOP/s See **Floating-Point Operation per Second**.

Flux In numerical analysis, particularly when solving conservation laws and partial differential equations, it represents the quantity that flows through a surface or across the boundaries of elements. In addition, numerical flux is the approximate flux at the interface between adjacent elements. See Equation (A.18), Appendix B.4.3, and Section 4.4.3.

Forward Problem Generate synthetic seismograms based on the defined 3D Earth model; see Section 2.1.3

FPGA See **Field-Programmable Gate Array**.

Full Element From the perspective of a particular element, a neighboring element is said to be a full element if its size is equal to that of a particular element. All elements are full-element for uniform mesh since they are equal in size.

Full-Wavefield Inversion A powerful method for obtaining high-resolution subsurface models by iteratively refining a subsurface model to generate synthetic seismic data that closely resembles the actual observed seismic data. It requires solving the wave equation multiple times, making it computationally expensive. See Section 1.1.

FWI See **Full-Wavefield Inversion**.

Gauss-Jacobi-Lobatto A numerical integration technique that extends the idea of Gaussian quadrature by incorporating specific boundary points of the integration interval as part of the quadrature nodes. The GJL quadrature is used by Hesthaven and Warburton (2010) to overcome the node clustering phenomenon of GLL quadrature, which is used in this dissertation. See Section 2.3.2.

Gauss-Legendre A numerical integration technique to approximate the integral of a function by selecting specific points (i.e., nodes) using the roots of Legendre polynomials and corresponding weights within an interval. See Section 2.3.2.

Gauss-Lobatto-Legendre A numerical integration technique that collocates the quadrature nodes and the finite element nodes, decoupling the derivative computations in one direction from solution values in another direction. See Section 2.3.2.

GDDR See **Graphics Dual-Data Rate**.

GEMM See **General Matrix-Matrix Multiplication**.

General Matrix-Matrix Multiplication Fundamental operation in numerical linear algebra to compute the product of two matrices with optional scaling and accumulation. It is a common operation in scientific computing and machine learning and is part of BLAS.

Ghost Buffer A buffer used to stage data of the mirror elements before sending them to appropriate MPI processes during ghost exchange. See Section 4.5.1.2.

Ghost Element Element that lives inside the memory of other MPI processes and is required by local element when computing flux. The MPI process stores the ghost element data inside the ghost layer, updated during ghost exchange. Instead of accessing the data directly from other MPI processes, the flux computation accesses the ghost element data from the ghost layer. See Section 4.5.

Ghost Exchange The process of exchanging data of ghost elements across all MPI processes through MPI communication. In the context of this dissertation, ghost exchange must be successfully finished before executing the flux kernel. See Section 4.5.

Ghost Layer A buffer used to receive the ghost elements from other MPI processes after ghost exchange is performed. See Section 4.5.1.1.

GJL See **Gauss-Jacobi-Lobatto**.

GL See **Gauss-Legendre**.

GLL See **Gauss-Lobatto-Legendre**.

Global Memory A type of GPU memory from a logical view whose memory space is visible to all threads inside the kernel grid. See Section 2.6.3.1.

GPU See **Graphics Processing Unit**.

GPUDirect A family of NVIDIA technologies that allows direct data exchange between GPUs and other devices. It includes GPUDirect Storage (i.e., between GPU and local/remote storage), GPUDirect RDMA (i.e., between local GPU and remote GPU), GPUDirect P2P (i.e., between local GPUs through the fastest fabric), and GPUDirect Video (i.e., efficient transfer of video frames from/to GPU memory). See Section 2.4.2.4.

Gradient A mathematical operator applied to a scalar function, which results in a vector field that points in the direction of the steepest ascent of that function at each point in the problem domain.

Graphics Dual-Data Rate A DRAM chip specifically designed to deliver extremely high data transfer rates compared to the standard (mainstream) DRAM. It provides high memory bandwidth for the graphics processing unit and is closely integrated with the GPU chip.

Graphics Processing Unit A specialized hardware that was historically designed to accelerate complex graphical tasks for visual applications. It features a massively-parallel execution engine to harness the parallel operations inherent in graphics applications. Nowadays, GPUs accelerate non-graphics parallel applications, taking advantage of their massive parallel execution to significantly speed up the applications against CPUs. See Section 2.6.

Grid Collection of thread blocks of a kernel launched on the GPU. See Table 2.1. A grid can contain as many as $2^{31} - 1$ thread blocks, each containing up to 1024 threads. Therefore, theoretically, a grid can have as many as 2 trillion threads.

Half Precision In computation, it refers to using a 16-bit floating-point format to store and perform arithmetic operations on real numbers. It has lower precision than single precision but is used in many machine-learning applications. Specialized functional units inside the hardware, such as the Tensor Cores in NVIDIA GPU, provide significant compute throughput when performing matrix multiplications in half precision.

Hanging Element From the perspective of a particular element, a neighboring element is said to be a hanging element if its size is smaller than that of a particular element, resulting in the nodes of the hanging element appearing to "hang" on the edge of that particular element. The hanging nodes only belong to one element since they do not coincide with the nodes of the adjacent element. This situation occurs in the non-conforming mesh. See Figure 2.3.

HBM See **High-Bandwidth Memory**

HDR High-Data Rate, which is an InfiniBand standard released in 2018 that provides 200 Gbps bandwidth in each direction.

Hexahedral Elements Three-dimensional elements that have eight vertices, twelve edges, and six quadrilateral faces.

High-Bandwidth Memory A memory technology that uses 3D-stacked DRAM chips to deliver extremely high data transfer rates compared to the standard (mainstream) DRAM and GDDR DRAM. It has very wide channels and is placed in the same package as the GPU chip, connected through a silicon interposer instead of copper wires on a printed circuit board like the GDDR memory.

High-Performance Computing Advanced computing techniques aimed to solve complex and large-scale problems by aggregating computing power and memory of many compute nodes connected through high-bandwidth, low-latency computer networks collectively called supercomputers or HPC clusters.

Host Memory A memory owned by the host processor (i.e., CPUs). In the context of this dissertation, it is a memory owned by the CPU that is not accessible by GPUs. Before running a kernel, necessary data must be copied from the host memory (i.e., CPU memory) to the device memory (i.e., GPU memory). At the end of kernel execution, the result is copied from the device memory to the host memory. NVIDIA provides Unified Virtual Memory (UVM) to automate this process, where the driver handles the data copy between host and device memories.

HPC See **High-Performance Computing**.

H-Tree In the context of interconnection, it is a hierarchical topology to connect all endpoints (e.g., components), resembling the structure of a tree. The lowest level of the tree connects a group of endpoints through a leaf switch. Then, a second-level switch connects a group of leaf switches, and so on. The levels depend on the total number of endpoints and the number of endpoints per leaf switch. Multiple data transfers between multiple endpoints can be performed simultaneously if they do not use the same switch. H-tree is one of the proposed interconnects for processing in memory, as described in Section 7.1.3.2.

Hydrophones A device that detects and records underwater sounds by measuring changes in water pressure. In other words, it is a microphone designed for underwater purposes. It is used during seismic surveys to record the reflected seismic waves, providing data about the subsurface geological formations.

Hyperbolic Partial Differential Equations A class of partial differential equations whose solutions describe wave propagation (e.g., sound waves, electro-

magnetic waves, and seismic waves). These equations model systems whose information propagates with finite speed and require initial conditions.

IFIS Infinity Fabric Inter-Socket, a communication interface between AMD CPUs that was introduced in 2017 to replace the older HyperTransport Technology. It achieves a transfer speed of 10.67 GT/s, 18 GT/s, and 32 GT/s on Naples, Rome, and Genoa CPUs. In addition, it is also used as a communication interface between AMD GPUs, such as the CDNA, CDNA2, and CDNA3 GPUs.

InfiniBand A channel-based computer networking standard promoted by InfiniBand Trade Association (IBTA) is commonly used in high-performance computing clusters to provide high throughput and low latency communication links between compute nodes.

Instruction-Set Architecture An interface between a computer's hardware and software that acts as a contract, defining the set of instructions a processor can execute, the format of these instructions, and the hardware operations they represent. It determines how software interacts with hardware and is foundational to processor design. Although the hardware implementation may differ, as long as the hardware designer obeys the ISA, software compiled to the ISA can run on the hardware.

Inter-device In the context of this dissertation, it is communication or data movement between devices (e.g., CPUs, GPUs), either inside the same compute nodes or different compute nodes. Intra-node interfaces will be used for communication between devices within the same compute nodes, while inter-node interfaces will be used for communication between devices located on different compute nodes. Note that using inter-node interfaces implicitly implies intra-node interfaces are also used (e.g., between GPUs and NICs).

Inter-node In the context of this dissertation, it is communication or data movement between devices (e.g., CPUs, GPUs) located in two or more compute nodes.

The communication uses a computer network (e.g., InfiniBand, Ethernet) that provides connection between nodes, which is often the weakest link (i.e., having the lowest bandwidth) in HPC clusters.

Intra-device In the context of this dissertation, it is communication or data movement within the device itself, most notably between on-chip memory (i.e., SRAM) and off-chip memory (i.e., DRAM).

Intra-node In the context of this dissertation, it is communication or data movement between devices (e.g., CPUs, GPUs) located within the same compute nodes. The communication uses high bandwidth interfaces that connect the devices, such as PCI Express and NVLink.

Inverse Problem Finding the optimum model of the Earth that best describes the data from seismic surveys; see Section 2.1.3.

IPMI Intelligent Platform Management Interface, which is a set of standard computer interface specifications for remote management and monitoring of the host computer.

ISA See **Instruction-Set Architecture**.

Iterative A procedure started with an initial guess of the solution, which is repeatedly refined through a series of calculations (i.e., iterations) to get closer to the final solution. The i-th solution is derived from $(i-1)$-th solution.

Kernel In numerical analysis, a kernel is the core computational routines or algorithms that drive the simulation of a physical, mathematical, or computational model. It is the "engine" that performs the heavy numerical computations required to solve the underlying equations describing the modeled system. In the context of GPU, a kernel is a small program consisting of a stream of instructions launched into the GPU (i.e., device) by the CPU (i.e., host) for data-parallel and thread-parallel execution.

Kernel Fusion Optimization strategy to merge two or more kernels into a single long kernel, reducing kernel launch and data movement overhead. Since the state of the GPU is reset, a subsequent kernel that needs data from the previous kernel must bring the data back from off-chip memory to the on-chip memory. By fusing kernels, the state of the GPU is preserved longer, and the data most likely still reside inside on-chip memory (e.g., caches), reducing the data movement between on-chip and off-chip memory. See Sections 5.4.1 and 6.2.

L_2 **Error** Sum of squared error between the final (computed) and analytical solutions. This error is used to verify the result of simulations when developing GPU codes. See Section 4.6.1.

Lamé parameters Two material properties denoted by λ and μ that arise from strain-stress relationship. The former is referred to as Lamé's first parameter, while the latter is referred to as Lamé's second parameter. See Section 2.2.2.

Local Memory A type of GPU memory from a logical view whose memory space is private to each thread. See Section 2.6.3.1.

Locality In computers, it is a tendency of an application to access the same set of memory addresses repetitively over a short period. Applications that exhibit locality can improve their performance using hardware features, such as caches, prefetchers, and branch predictors. There are two types of locality: temporal locality, which is the tendency of a program to access the same data multiple times in short intervals, and spatial locality, which is the tendency of a program to access data stored close to each other.

Lock-Step Execution The execution mechanism in GPU where all threads in a warp must execute the same instructions at the same time simultaneously. Any divergence in the execution path will reduce parallel execution efficiency since the threads must be serialized.

Logical Memory In the context of GPU, it is the memory hierarchy of GPU from software perspective, which includes the memory scope among threads, warps, and blocks, the allocation lifetime, and the type of data that can be stored. See Section 2.6.3.1.

Look-up Table A pre-stored set of data that allows for quick retrieval of an output value based on a given input value. It replaces complex operations with simpler array indexing operations. See Figures 5.1 and 5.5.

Low-Storage Runge-Kutta A variant of Runge-Kutta methods optimized to use minimal memory, making it especially suitable for large-scale simulations in computational science and engineering. See Section 2.3.3.

LSRK See **Low-Storage Runge-Kutta**.

LUT See **Look-up Table**.

`mass_inverse` An array with a length of `NNODE` inside the element data structure that stores the inverse of the diagonal mass matrix for each node in an element. See Section 4.2.3 as well as Appendix B.1.

`materials` An array with a length of `NUM_MATERIALS`×`NNODE_MATERIALS` inside the element data structure that stores the material properties for each node within an element. See Section 4.2.3 as well as Appendix B.1.

Memory Block The basic units of a PIM chip where the computation is performed. It contains memristor cells organized in a 2D array and peripheral components, including sense amplifiers, decoders, row drivers, column drivers, row buffers, and column buffers. In the context of this dissertation, one memory block consists of 1024×1024 memristor cells, resulting in 1 Mbit capacity. See Section 7.1.1.

Memory Bound An application whose performance is primarily limited by the speed of memory accesses (i.e., memory bandwidth).

Memory Tile A collection of memory blocks inside a PIM chip. In the context of this dissertation, one memory tile consists of 16×16 memory blocks. The capacity of the PIM chip can be controlled by varying the number of memory tiles inside the chip. See Section 7.1.1.

Memristor A fundamental electronic component that combines memory and resistance in a single device. It acts as non-volatile memory, retaining information even when power is removed, and its resistance depends on the history of the voltage applied and the current that has passed through it. See Section 2.7.2.

Mesh In numerical analysis, a mesh is a discretized representation of a computational domain used to solve partial differential equations (PDEs) or perform numerical simulations. It divides the domain into smaller, simpler elements (e.g., triangles, quadrilaterals, tetrahedra, or hexahedra), facilitating the approximation of complex problems through numerical methods.

Message passing A method of communication between different computing nodes or processes in parallel computing systems, where data is exchanged by sending and receiving "messages" without relying on shared memory space, making it ideal for distributed computing environments. Each node or process can work on separate parts of a problem simultaneously while coordinating their results through these message exchanges. See Section 2.4.2.

Message Passing Interface A standard maintained by the Message Passing Interface (MPI) Forum to provide standards on semantic, syntax, and low-level routines of the message passing paradigm, allowing for wide portability. See Section 2.4.2.

Mirror Element Elements that become ghost elements for other MPI processes. In other words, local elements that are ghosts in the perspective of at least one MPI process.

Mixed Precision A method for computing floating-point numbers by using different precision formats: lower precision, such as half-precision for parts of the computation that do not significantly impact numerical accuracy, and higher precision, such as double precision for critical parts of the calculation to ensure numerical stability and accuracy. See Section 2.6.5.

MPI See **Message Passing Interface**.

`ncu` Nsight Compute CLI, a non-interactive profiler by NVIDIA Corporation (2023a), to collect metrics of applications running on NVIDIA GPU. The metrics can be used to study the behavior of the applications, their interaction with hardware, and, ultimately, identify the key bottleneck for further optimizations. NCU supports NVIDIA Volta GPU or newer generations. For older GPU, use `nvprof`.

Near-Memory Processing One approach of compute-in-memory which brings the compute units near the memory arrays and integrates them at chip or package levels. In other words, NMP still relies on separate compute units placed near the memory arrays. See Section 2.7.

Network Interface Card A hardware component installed in a computer that allows it to connect to a network following a particular standard (e.g., InfiniBand, Ethernet). It acts as a bridge between the computer and the network, enabling data transmission and communication with other devices on the network. Modern NICs use the PCI Express interface to communicate with the rest of the hardware within a computer.

NIC See **Network Interface Card**.

NMP See **Near-Memory Processing**.

`NNODE` In the context of this dissertation, it denotes the number of nodes in each element. In 3D space, $\texttt{NNODE} = \texttt{NNODE_1D}^3$. See Section 4.1.2.

`NNODE_1D` In the context of this dissertation, it denotes the number of nodes in each direction for an element. See Section 4.1.2.

`NNODE_FACE` In the context of this dissertation, it denotes the number of nodes in each face of an element. In 3D space, `NNODE` = `NNODE_1D`^{2}. See Section 4.1.2.

`NNODE_MATERIALS` Number of material nodes in each direction, which can be constant, linear, or the same as `NNODE_1D`. Only constant material is considered in this dissertation. See Appendix B.5.3.

Nodes In the context of numerical analysis, see **Element Node**. In the context of the HPC cluster, see **Compute Node**.

Non-Blocking Communication Communication functions in MPI where the thread only issues the sending/receiving request and can continue its execution flow. See Section 2.4.2.

Non-conforming Mesh A type of computational mesh used in numerical simulation where the nodes between adjacent elements do not perfectly align. In other words, it has hanging nodes, allowing more flexibility in meshing complex geometries by enabling different mesh densities in different areas. However, special techniques are required to ensure accurate analysis solutions.

Non-Volatile Memory A type of computer memory that can retain data even when the power is off. The non-volatile memory technologies include flash-based NVM (e.g., NAND Flash, NOR Flash) and phase-change NVM.

`NUM_MATERIALS` Number of material properties each node has. There are two and three material properties for acoustic and elastic wave simulations, respectively. See Section 4.2.3.

`NUM_VARS` In the context of this dissertation, it denotes the number of variables being solved for each node in each element. There are four and nine variables for

acoustic and elastic wave simulations in 3D space, respectively. See Section 4.2.3 as well as Appendix B.1.

NVIDIA-SMI A system management interface provided by NVIDIA Corporation (2020c) as a command line tool that allows users to monitor, query, and configure NVIDIA GPUs.

NVLink A wire-based, serial multi-lane, near-range proprietary communication link developed by NVIDIA, providing high-bandwidth, energy-efficient communication interfaces between NVIDIA GPUs. In addition, the IBM POWER9 CPU uses NVLink to connect to the GPU instead of the commonly-used PCI Express interface.

NVM See **Non-Volatile Memory**.

`nvprof` NVIDIA Profiling Tool, a non-interactive profiler by NVIDIA Corporation (2024), to collect metrics of applications running on NVIDIA GPU. The metrics can be used to study the behavior of the applications, their interaction with hardware, and, ultimately, identify the key bottleneck for further optimizations. The `nvprof` supports NVIDIA Turing GPU or older generations. For newer GPU, use `ncu`.

NVSim Work by Dong et al. (2012) from which the model parameters of a non-volatile memory circuit are obtained to model latency and area of memristor-based PIM in this dissertation. See Section 3.3.2.

Octant A terminology used by `p4est` to call hexahedral elements in 3D space. See Section 2.4.1.

Octree A tree-based data structure in which each internal node has exactly eight children. It represents a hierarchical subdivision of 3D space into eight octants, beneficial for creating and managing adaptive meshes, where the resolution of

the mesh can vary depending on the complexity of the solution or geometry in different parts of the domain.

Off-chip Memory A memory located outside the compute chip. DRAM is the most common off-chip memory for CPUs and GPUs. The HBM on GPU is still considered off-chip memory, although it is placed in the same package as the GPU chip.

On-chip Memory A memory located inside the compute chip, and thus, near the ALUs. SRAM is the most common on-chip memory for CPUs and GPUs. It is either hardware-managed (e.g., registers, caches) or user-managed (e.g., shared memory, scratchpad) memory.

One-sided Communication An MPI communication pattern where only one process initiates the data transfer, allowing it to directly access and modify the memory of another process without involving explicit coordination from another process. This contrasts with point-to-point communication, where the sender and receiver must call a send and receive command, respectively. See Section 2.4.2.2.

Open Computing Library An open, cross-platform framework maintained by Khronos Group for writing programs that execute across heterogeneous platforms, such as CPUs and GPUs. It provides a unified programming model to efficiently utilize diverse hardware devices' computational power efficiently, acting as API for heterogeneous computing.

OpenCL See **Open Computing Library**.

Operator Overloading A feature in some programming languages that allows users to redefine how the operators (e.g., $+$, $-$, $*$, $==$) work for user-defined types. This enables objects of custom classes or structures to interact using standard operators in an intuitive way that aligns with their intended behavior.

p4est A software library by Burstedde et al. (2011); Isaac et al. (2015, 2012) that allows parallel dynamic management of a collection of adaptive octress (i.e., forest of octress). It is the adaptive mesh refinement library (AMR) used in this dissertation. See Section 2.4.1.

Partial Differential Equation A type of mathematical equation that involves unknown functions of multiple variables and their partial derivatives. It describes a wide range of physical, biological, and engineering systems: steady-state problems using elliptic PDEs, diffusion-like processes using parabolic PDEs, and wave-like phenomena using hyperbolic PDEs.

PCI Express See **Peripheral Component Interconnect Express**.

PCIe See **Peripheral Component Interconnect Express**.

PDE See **Partial Differential Equation**.

Peripheral Component Interconnect Express A high-speed, point-to-point communication protocol and interface standard used for connecting peripheral devices inside a computer. It is widely employed in modern computing for components such as graphics processing units (GPUs), solid-state drives (SSDs), and network interface cards (NICs).

Physical Memory In the context of GPU, it is the actual memory hierarchy of GPU implemented in the hardware. It consists of both hardware-managed on-chip, user-managed on-chip, and off-chip memories. See Section 2.6.3.2.

PIM See **Processing-in-Memory**.

Pipelining A technique of designing processor architecture by dividing execution into smaller stages, each handled by a specific part of the processor. It resembles the production line in a factory, where instructions flow through each stage, allowing multiple instructions to be in different stages of execution simultaneously, improving instruction-level parallelism.

Point-to-Point Communication An MPI communication pattern where one specific sender MPI process transfers a message to a particular receiver MPI process, both residing in the same communicator. The sender must execute the send command, while the receiver must execute the receive command with the same source/destination.

POWER9 A family of superscalar, multithreading, multi-core microprocessors that was announced in 2016 and launched in 2017 by IBM.

Process Node A specific manufacturing technology or fabrication process used to create semiconductor devices, such as CPUs, GPUs, and memory chips. The process node is often associated with the smallest feature size or transistor dimensions achievable during the production of integrated circuits. Smaller process nodes increase transistor density, reduce power consumption, and allow transistors to switch faster, improving performance.

Processing-in-Memory One approach of compute-in-memory which performs the computation directly in the memory arrays, leveraging the properties of the materials used to construct the arrays. In other words, unlike NMP, PIM does not rely on separate compute units. See Section 2.7.

QPI Quick-Path Interconnect, a communication interface between Intel CPUs (i.e., inter-socket communication), reaching 6.4 GT/s, 8 GT/s, and 9.6 GT/s for Nehalem EP, SandyBridge EP, and Haswell EP CPUs, respectively. It was replaced by Ultra-Path Interconnect in 2017.

Quadrant A terminology used by `p4est` to call quadrilateral elements in 2D space. See Section 2.4.1.

Quadrature In the context of numerical analysis, it refers to approximating a function's definite integral, typically over a finite interval.

Quadrilateral Two-dimensional elements in the form of a four-sided polygon with four edges and four corners.

Quadtree A tree-based data structure in which each internal node has exactly four children (i.e., a two-dimensional analog of octrees). It represents a hierarchical subdivision of 2D space into four quadrants, which is beneficial for creating and managing adaptive meshes.

Race Condition A situation in computing where the application's behavior depends on the sequence or timing of uncontrollable events, such as the order in which threads or processes execute when they access shared resources simultaneously (e.g., data) and the outcome depends on the timing of their execution. It can lead to unpredictable behavior, incorrect results, or even system crashes.

Rank In the context of MPI communication, it refers to a unique integer identifier assigned to each MPI process within a communicator, allowing processes to distinguish themselves from one another and facilitating communication by providing a numerical label for each process.

RAPL Running Average Power Limit, a feature on Intel processors that allows for real-time monitoring and management of CPU and RAM power consumption. See explanation by David et al. (2010).

RDMA See **Remote Direct Memory Access**.

RDMA-over-Converged Ethernet A network protocol that adds RDMA on commodity Ethernet. The basic form of Ethernet does not support RDMA and does not guarantee the arrival and latency of traffic packets. With priority flow control, RoCE achieve lossless and guaranteed latency on traffic packets over commodity Ethernet with support for RDMA. See explanation by Hanindhito et al. (2024).

Refinement Level Controls the refinement of the mesh used for simulation; a higher level produces finer mesh consisting of a higher number of smaller elements. See Section 4.1.1.

Reflection Seismology A geophysical method used to explore and map the Earth's subsurface by analyzing the seismic waves' reflection from geological layers. It is widely used in oil and gas exploration, mineral exploration, and geotechnical investigations. Seismic waves are generated and sent into the ground. When these waves encounter interfaces between materials with different acoustic properties (e.g., different rock types or fluids), some waves are reflected to the surface. The time it takes for the waves to return to the surface is recorded, and from this data, a profile of the subsurface layers is constructed. See Section 2.1.

Register Files A collection of registers, which are small-capacity, high-speed on-chip memory near the ALUs inside the CPU. It allows for direct access to the data used during calculations.

Register Spill A condition when a thread has a large amount of data (e.g., operand, intermediate results) that cannot be stored inside the physical registers in a processor due to insufficient capacity, making some data spill over to local memory, which can be significantly slower than the registers.

Remote Direct Memory Access A technology that enables the direct transfer of data between the memory of two computers in a network without involving their operating systems (OS) or processors (CPU). With RDMA-capable NIC, the NIC can directly transmit or receive the data from or to the CPU memory without explicitly staging the data inside the NIC's buffer.

Remote Memory Access A feature of MPI that allows one process to directly access the memory of another process in a distributed memory system. This feature provides a programming model where processes can perform one-sided communication: a process can read from or write to another process's memory

without the active involvement of the remote process. This is not to be confused with remote direct memory access.

RMA See **Remote Memory Access**.

RoCE See **RDMA-over-Converged Ethernet**.

Roofline Analysis Analysis technique on the behavior of an application by plotting it into the roofline chart, providing an intuitive and insightful way to compare application performance against machine capabilities. Roofline analysis will help determine whether an application is compute-bound or memory-bound on that particular hardware. See Section 2.5.2.

Roofline Chart A log-log plot where the y-axis represents the compute throughput measured in GFLOP/s and the axis represents the arithmetic intensity measured in FLOP/byte. See Section 2.5.2.

Roofline Model Model of a particular computing hardware drawn into a roofline chart as the roof and slope lines. The roof represents the peak compute throughput the hardware can perform on specific data types (i.e., arithmetic precision). The slope represents the peak memory bandwidth for a specific memory type. See Section 2.5.2.

Row Buffer Temporary storage inside memory block to store recently-read row, allowing fast access to data if the subsequent accesses go into the same row.

Row Driver Circuit to select a specific row within the memory array, activate it, and copy the data into the row buffer, allowing data to be read or written.

Row-Parallel Execution A parallel execution inside a memory block where all rows perform the same operations but with different data, assuming different data is stored in each row. It can be viewed as a single-instruction multiple-data for the memory block. See Section 2.7.2.

Runge-Kutta A family of numerical methods used to solve initial-value problems for differential equations. It approximates the solution by advancing discrete time steps from one point to the next, improving accuracy using multiple intermediate evaluations of the derivative function within each step. See Section 2.3.3.

Seismic Survey A technique used to investigate and map the subsurface of the Earth by sending seismic waves into the ground and analyzing how these waves reflect, refract, or travel through different underground materials. It is commonly used in geological exploration, oil and gas exploration, mining, and civil engineering. See Section 2.1.

Seismic Waves Waves that travel through the Earth's layers or along its surface, typically generated by earthquakes, volcanic activity, landslides, or man-made explosions. They allow scientists to study the planet's interior structure by analyzing their travel times and patterns. See Section 2.1.

Seismogram Output of seismograph that visually records ground motion caused by seismic waves.

Seismograph An instrument used to detect and record seismic waves, allowing scientists to measure the intensity and location of the source by recording the ground motion caused by these waves.

SEM See **Spectral Element Method**.

Sense Amplifier A circuit designed to detect and amplify very small voltage signals stored in memory cells, allowing other circuits to accurately read the data stored as a '1' or '0'. In other words, it boosts the weak signal to a recognizable logic level that can be interpreted by other circuits.

Shape Function In the context of numerical analysis, it is a mathematical function used to approximate the variation of physical quantity within an element. See Appendix A.4.2.

Shared Memory A user-managed, on-chip memory inside a GPU that stores repeatedly used data and shares data between threads within a thread block. See Section 2.6.3.2.

SIMD See **Single Instruction Multiple Data**.

SIMT See **Single Instruction Multiple Thread**.

Single Instruction Multiple Data A computing technique where an instruction is executed to operate on multiple data simultaneously, allowing for parallel processing and significantly improving performance for tasks that involve repetitive operations on large datasets. In other words, it applies the same operation to multiple data simultaneously with just one instruction.

Single Instruction Multiple Thread A modification to SIMD, allowing the execution of one instruction into multiple independent threads in parallel. This allows programmers to develop thread-parallel code for individual threads (i.e., SIMT) and data-parallel code for coordinated threads (i.e., SIMD) on general-purpose GPU. However, each thread within a warp executes in lock-step fashion.

Single Precision In computation, it refers to using a 32-bit floating-point format to store and perform arithmetic operations on real numbers. It is sufficient for use in applications where extreme numerical precision is not required, such as computer graphics, machine learning, and gaming. Some scientific applications also prefer to use single precision for cases where precision is less critical since it has 50% lower memory for storage and 50% lower bandwidth for communication. In addition, hardware usually has twice the compute throughput for single precision compared to double precision.

SM See **Streaming Multiprocessor**.

SM Occupancy A metric that measures how effectively an SM inside the GPU is utilized by calculating the ratio of active warps that can be scheduled into an

SM compared to the maximum number of warps that can be concurrently active in an SM. Four factors affect the theoretical SM Occupancy: Warps per SM, Blocks per SM, Registers per SM, and Shared Memory per SM.

SM-Bus System Management Bus is a single-ended simple two-wire bus that connects system components for lightweight communication.

SMSP See **Streaming Multiprocessor Sub-Partition**.

Spatial Integration In numerical analysis, it is a process of approximating a definite integral over a spatial (space) domain, by calculating the sum of a function using a numerical method. Depending on the dimension of the problem domain, it can involve various forms of integration, including volume integrals, surface integrals, or line integrals.

Spectral Element Method A numerical method used to solve differential equations by combining the high accuracy of spectral methods with the geometric adaptability of finite elements. It uses high-degree piecewise polynomials as basis functions.

Spine-Leaf A network architecture commonly used in HPC to connect the compute nodes in a cluster using two switching layers: leaf switches and spine switches. The former consists of access switches that aggregate traffic from compute nodes and connect directly to the spine switches. The latter connects all leaf switches to a full-mesh topology. Note that a two-level fat tree can be considered as spine-leaf, and a three-level fat tree can be considered as super_spine-spine-leaf.

SRAM See **Static Random Access Memory**.

Static Random Access Memory A random access memory technology that uses bistable latching circuitry to store each bit of data. Unlike DRAM, which needs periodic refreshes, SRAM does not; it stores the data as long as power is supplied. While it is more expensive and has a lower density than DRAM, its

ability to provide low-latency access to data makes SRAM used as a CPU or GPU cache.

Streaming Multiprocessor A key architectural component of NVIDIA GPU responsible for executing multiple threads inside a thread block in parallel. It resembles a CPU core but with significantly many ALUs (i.e., CUDA Cores). It has a set of registers, shared memory, and caches and contains four sub-partitions (SMSP). The SM schedules warp to an SMSP for execution. See Section 2.6.2.

Streaming Multiprocessor Sub-Partition A part of an SM that is responsible for executing a warp of threads. It features various execution units (e.g., FP64 CUDA Cores, FP32 CUDA Cores, Special Function Units, Tensor Cores). With respect to the SIMT execution model of the GPU, an SMSP can be viewed as SIMT processor, executing all threads within a warp in lock-step fashion.

Structural Hazard A type of hazard in computing where two instructions request the same hardware resource simultaneously. In the case of memristor-based processing-in-memory used in this dissertation, structural hazard happens when data transfer in the memory block happens while the memory block performing parallel-operations.

Structured Mesh A type of computational grid used to discretize a problem domain for solving partial differential equations (PDEs). It consists of a regular, grid-like pattern that makes it easy to access adjacent elements and reduces computational costs. However, mesh refinement is more complex, especially for problem domains with sharp features.

TACC Texas Advanced Computing Center, an advanced computing research center at The University of Texas at Austin. It operates advanced high-performance computing clusters for researchers in Texas and across the U.S. The clusters include Stampede3, Lonestar6,

TC See **Tensor Cores**.

Temporal Integration In numerical analysis, it is a process to approximate the solution to differential equations over time by dividing the time domain into small intervals and calculating the change in the `variables` within each interval. In other words, it accumulates the changes of `variables` to observe how the system behaves over time, which is essential for solving time-dependent problems.

Tensor Cores Specialized functional units inside an SMSP in NVIDIA GPU to accelerate GEMM operations with support for many lower precision data types, mainly to accelerate machine learning applications. It is generally called a matrix accelerator. See Table 2.1.

Texture Cache A special, hardware-managed, read-only on-chip memory that stores part of image data (e.g., texture) used for putting images into polygons/triangles (i.e., texture mapping).

Thread Single stream of instruction and data with its register allocation and local memory spaces. See Table 2.1.

Thread Block Terminology in NVIDIA GPU that denotes a group of warps consisting of up to 1024 threads. A thread block is scheduled for execution into one SM. See Table 2.1.

Thread Divergence In the context of parallel execution in GPU, it is a condition where multiple threads within a group (i.e., warp) take different execution paths due to conditional branching. Since each thread within a warp must be executed in lock-step fashion, thread divergence reduces the efficiency of parallel execution due to execution serialization for each path: the GPU will execute threads with one execution path followed by threads taking another execution path.

Tiling An optimization technique where large data is divided into smaller "tiles" to improve memory access efficiency by maximizing caching locality. See Section 6.1.

Type Cast In the context of programming, it is a process of converting a value from one data type to another, which is essentially changes how the computer interprets and stores the value.

Unified Shaders A hardware structure inside a modern GPU that can run all vertex, fragment, and geometry tasks without distinction, depending on the kernel that is being executed. Previously, each task had its specialized hardware structure. Unified shaders mark the beginning of general-purpose GPU, opening the possibility to run non-graphics kernels.

Uniform Mesh A computational mesh where all elements have equal size. See Figure 2.3. This dissertation only considers uniform mesh.

Unstructured Mesh A type of computational grid used to discretize a problem domain for solving partial differential equations (PDEs). It consists of irregularly shaped and distributed elements, allowing it to conform to complex geometries and capture localized features more efficiently at the expense of higher computational costs.

UPI Ultra-Path Interconnect, a communication interface between Intel CPUs (i.e., inter-socket communication), reaching 10.4 GT/s and 11.2 GT/s transfer speed per link for Intel Skylake SP and Icelake SP CPUs, respectively. It was first introduced with Skylake-SP processors in 2017 and replaced Quick-Path Interconnect. Second-generation UPI was introduced with Sapphire Rapids CPU, reaching a transfer speed of 16 GT/s per link.

`variables` An array with a length of `NNODE`×`NUM_VARS` inside the element data structure that stores the unknown values which must be evaluated for each time step. See Section 4.2.3 as well as Appendix B.1.

Volume In numerical analysis, volume refer to the operations done inside the interior of an element. See Sections 4.4.2 and 5.3.2.

von-Neumann architecture A computer design model with a single system bus connecting the CPU, memory, and input/output devices. It stores the instructions (i.e., programs) in memory and then executes them sequentially. Both instructions and data are stored in the same memory (i.e., both instructions and data have the same address space).

von-Neumann bottleneck The limitation found in von-Neumann architecture that has CPU and memory as separate components, requiring data transfer between memory and CPU to perform computation. Since the CPU is significantly faster than the memory, it often waits for data, reducing overall system performance.

Warp Terminology in NVIDIA GPU that denotes a group of 32 threads that execute the same instruction stream in lock-step fashion and operate on different data. It is executed by an SMSP inside the SM. See Table 2.1.

Wavefront Terminology in AMD GPU that denotes a group of 32 threads that execute the same instruction stream in lock-step fashion and operate on different data. See Table 2.1.

World In the context of MPI Communication, it refers to the default communicator, `MPI_COMM_WORLD`, containing all of MPI processes. It is the initial communication context and serves as the starting point for MPI-based applications. See Section 2.4.2.2.

Zero Copy With RDMA, there is no need for intermediate copies to the intermediate buffers when transferring data between the local computer's memory and the remote computer's memory. Thus, it is called zero-copy data transfer.

References

Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. Novel HPC Techniques to Batch Execution of Many Variable Size BLAS Computations on GPUs. In *Proceedings of the International Conference on Supercomputing*, ICS '17, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350204.

Daniel S. Abdi, Lucas C. Wilcox, Timothy C. Warburton, and Francis X. Giraldo. A GPU-Accelerated Continuous and Discontinuous Galerkin Non-Hydrostatic Atmospheric Model. *International Journal of High Performance Computing Applications*, 33(1):81–109, 2019. ISSN 17412846.

Aria Abubakar, Gong Li Wang, Lin Liang, Tarek M. Habashy, and Maokun Li. Electromagnetic Modeling and Inversion Application for Oil and Gas Industry. In *2016 Progress in Electromagnetic Research Symposium (PIERS)*, pages 938–938, 2016.

Advanced Micro Devices. AMD CDNA Architecture. Whitepaper, Advanced Micro Devices, US, November 2020.

Advanced Micro Devices. AMD CDNA™ 3 Architecture. The All-New AMD GPU Architecture for the Modern Era of HPC and AI. Whitepaper, Advanced Micro Devices, California, US, January 2023.

Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing. *SIGARCH Comput. Archit. News*, 43(3S):105–117, jun 2015a. ISSN 0163-5964.

Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 336–348, New York, NY, USA, 2015b. Association for Computing Machinery. ISBN 9781450334020.

James Ahrens, Berk Geveci, and Charles Law. ParaView: An End-User Tool for Large Data Visualization. In *Visualization Handbook*. Elesvier, 2005. ISBN 978-0123875822.

K. Anandhanarayanan, Konark Arora, Vaibhav Shah, R. Krishnamurthy, and Debasis Chakraborty. Separation Dynamics of Air-to-Air Missile using a Grid-Free Euler Solver. *Journal of Aircraft*, 50(3):725–731, 2013.

Shaahin Angizi, Zhezhi He, Dayane Reis, Xiaobo Sharon Hu, Wilman Tsai, Shy Jay Lin, and Deliang Fan. Accelerating Deep Neural Networks in Processing-in-Memory Platforms: Analog or Digital Approach? In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 197–202, 2019.

Aayush Ankit, Izzat El Hajj, Sai Rahul Chalamalasetti, Geoffrey Ndu, Martin Foltin, R. Stanley Williams, Paolo Faraboschi, Wen-mei W Hwu, John Paul Strachan, Kaushik Roy, and Dejan S. Milojicic. PUMA: A Programmable Ultra-efficient Memristor-based Accelerator for Machine Learning Inference. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 715–731, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362405.

Daniel Appelö and Thomas Hagstrom. An Energy-Based Discontinuous Galerkin Discretization of The Elastic Wave Equation in Second Order Form. *Computer Methods in Applied Mechanics and Engineering*, 338:362–391, 2018. ISSN 0045-7825.

R. Archibald, K.J. Evans, and A. Salinger. Accelerating Time Integration for The Shallow Water Equations on The Sphere Using GPUs. *Procedia Computer Science*, 51:2046–2055, 2015. ISSN 18770509.

Kazi Asifuzzaman, Mohammad Alaul Haque Monil, Frank Liu, and Jeffrey S Vetter. Evaluating HPC Kernels for Processing in Memory. In *Proceedings of the 2022 International Symposium on Memory Systems*, MEMSYS '22, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450398008.

Scott Atchley, Christopher Zimmer, John Lange, David Bernholdt, Veronica Melesse Vergara, Thomas Beck, Michael Brim, Reuben Budiardja, Sunita Chandrasekaran, Markus Eisenbach, Thomas Evans, Matthew Ezell, Nicholas Frontiere, Antigoni Georgiadou, Joe Glenski, Philipp Grete, Steven Hamilton, John Holmen, Axel Huebl, Daniel Jacobson, Wayne Joubert, Kim Mcmahon, Elia Merzari, Stan Moore, Andrew Myers, Stephen Nichols, Sarp Oral, Thomas Papatheodore, Danny Perez, David M. Rogers, Evan Schneider, Jean-Luc Vay, and P. K. Yeung. Frontier: Exploring Exascale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701092.

Utkarsh Ayachit, Andrew Bauer, Berk Geveci, Patrick O'Leary, Kenneth Moreland, Nathan Fabian, and Jeffrey Mauldin. ParaView Catalyst: Enabling In Situ Data Analysis and Visualization. In *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV 2015)*, pages 25–29, November 2015.

Abdelkader Baggag, Harold Atkins, and David Keyes. *Parallel Implementation of The Discontinuous Galerkin Method*, chapter 11. Elsevier Science, 2000. ISBN 9780080538389.

C.C. Bates, T.F. Gaskell, and R.B. Rice. *Geophysics in the Affairs of Man: A Personalized History of Exploration Geophysics and Its Allied Sciences of Seismology and Oceanography*. Pergamon international library of science, technology, engineering, and social studies. Elsevier Science, 2016. ISBN 9781483152219.

Himanshu Bhatnagar. *Advanced ASIC Chip Synthesis*. Springer, 2002. ISBN 0792376447.

Ansel L. Blumers, Yu-Hang Tang, Zhen Li, Xuejin Li, and George E. Karniadakis. GPU-Accelerated Red Blood Cells Simulations with Transport Dissipative Particle Dynamics. *Computer Physics Communications*, 217:171–179, aug 2017. ISSN 00104655.

David Blythe. Rise of the Graphics Processor. *Proceedings of the IEEE*, 96(5): 761–778, 2008.

Cade Brown, Ahmad Abdelfattah, Stanimire Tomov, and Jack Dongarra. Design, Optimization, and Benchmarking of Dense Linear Algebra Algorithms on AMD GPUs. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, Virtual, 2020. Institute of Electrical and Electronics Engineers.

Nicolas Brunie, Caroline Collange, and Gregory Diamos. Simultaneous Branch and Warp Interweaving for Sustained GPU Performance. *SIGARCH Comput. Archit. News*, 40(3):49–60, June 2012. ISSN 0163-5964.

Ian Buck. GPU Computing: Programming a Massively Parallel Processor. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, page 17, USA, 2007a. IEEE Computer Society. ISBN 0769527647.

Ian Buck. GPU Computing with NVIDIA CUDA. In *ACM SIGGRAPH 2007 Courses*, SIGGRAPH '07, page 6–es, New York, NY, USA, 2007b. Association for Computing Machinery. ISBN 9781450318235.

Nicholas Buoncristiani, Sanjana Shah, David Donofrio, and John Shalf. Evaluating the numerical stability of posit arithmetic. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 612–621, 2020.

Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. p4est: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, 2011.

J.C. Butcher and G. Wanner. Runge-Kutta Methods: Some Historical Notes. *Applied Numerical Mathematics*, 22(1):113–151, 1996. ISSN 0168-9274. Special Issue Celebrating the Centenary of Runge-Kutta Methods.

Yi Cai, Tianqi Tang, Lixue Xia, Ming Cheng, Zhenhua Zhu, Yu Wang, and Huazhong Yang. Training Low Bit Width Convolutional Neural Network on RRAM. In *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 117–122, 2018.

M.H. Carpenter and C.A. Kennedy. Fourth-order 2N-storage Runge-Kutta Schemes. Technical Report 109112, NASA Langley Research Center, 1994.

Jesse Chan. Weight-Adjusted Discontinuous Galerkin Methods: Matrix-Valued Weights and Elastic Wave Propagation in Heterogeneous Media. *International Journal for Numerical Methods in Engineering*, 113(12):1779–1809, 2018.

Jesse Chan, Zheng Wang, Axel Modave, Jean Francois Remacle, and T. Warburton. GPU-Accelerated Discontinuous Galerkin Methods on Hybrid Meshes. *Journal of Computational Physics*, 318:142–168, 2016. ISSN 10902716.

M. V. K. Chari. Electromagnetic Modeling of Electrical Machinery for Design Applications. In J. Caldwell and R. Bradley, editors, *Industrial Electromagnetics Modelling*, pages 3–14, Dordrecht, 1983. Springer Netherlands. ISBN 978-94-009-6917-9.

Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A Performance Study of General-Purpose Applications on Graphics Processors using CUDA. *Journal of Parallel and Distributed Computing*, 68(10):1370–1380, 2008. ISSN 0743-7315. General-Purpose Processing using Graphics Processing Units.

Jian-Yu Chen, Fue-Sang Lien, Chong Peng, and Eugene Yee. GPU-Accelerated Smoothed Particle Hydrodynamics Modeling of Granular Flow. *Powder Technology*, 359:94–106, jan 2020. ISSN 00325910.

Jiefu Chen and Qing Huo Liu. Discontinuous Galerkin Time-Domain Methods for Multiscale Electromagnetic Simulations: A Review. *Proceedings of the IEEE*, 101(2):242–254, 2013.

Xuhang Chen, Xueyan Wang, Xiaotao Jia, Jianlei Yang, Gang Qu, and Weisheng Zhao. Accelerating Graph-Connected Component Computation with Emerging Processing-in-Memory Architecture. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(12):5333–5342, 2022.

Hui Cheng, Songbai Cheng, and Jiyun Zhao. Study on Corium Jet Breakup and Fragmentation in Sodium with a GPU-Accelerated Color-Gradient Lattice Boltzmann Solver. *International Journal of Multiphase Flow*, 126:103264, may 2020. ISSN 03019322.

John Runwei Cheng and Mitsuo Gen. Accelerating Genetic Algorithms with GPU Computing: A Selective Overview. *Computers & Industrial Engineering*, 128:514–525, feb 2019. ISSN 03608352.

Ming Cheng, Lixue Xia, Zhenhua Zhu, Yi Cai, Yuan Xie, Yu Wang, and Huazhong Yang. TIME: A Training-in-Memory Architecture for Memristor-Based Deep Neural Networks. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2017.

Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, page 27–39. IEEE Press, 2016. ISBN 9781467389471.

Jack Choquette, Olivier Giroux, and Denis Foley. Volta: Performance and Programmability. *IEEE Micro*, 38(2):42–52, 2018.

Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. NVIDIA A100 Tensor Core GPU: Performance and Innovation. *IEEE Micro*, 41(2):29–35, 2021.

Zamshed Chowdhury, Jonathan D. Harms, S. Karen Khatamifard, Masoud Zabihi, Yang Lv, Andrew P. Lyle, Sachin S. Sapatnekar, Ulya R. Karpuzcu, and Jian-Ping Wang. Efficient In-Memory Processing Using Spintronics. *IEEE Computer Architecture Letters*, 17(1):42–46, 2018.

Stefan Dan Ciocirlan, Dumitrel Loghin, Lavanya Ramapantulu, Nicolae Ţăpuş, and Yong Meng Teo. The Accuracy and Efficiency of Posit Arithmetic. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pages 83–87, 2021.

Bernardo Cockburn, George E. Karniadakis, and Chi-Wang Shu, editors. *Discontinuous Galerkin Methods: Theory, Computation, and Applications*. Springer Berlin Heidelberg, 2000. ISBN 9783642597213.

Guohao Dai, Tianhao Huang, Yuze Chi, Jishen Zhao, Guangyu Sun, Yongpan Liu, Yu Wang, Yuan Xie, and Huazhong Yang. GraphH: A Processing-in-Memory Architecture for Large-Scale Graph Processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(4):640–653, 2019.

Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. RAPL: Memory Power Estimation and Capping. In *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, pages 189–194, 2010.

David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A. Wood. Implementation techniques for main memory database systems. *SIGMOD Rec.*, 14(2):1–8, June 1984. ISSN 0163-5808.

Richard Diehl, Kurt Busch, and Jens Niegemann. Comparison of low-storage Runge-Kutta schemes for discontinuous Galerkin time-domain simulations of Maxwell's equations. *Journal of Computational and Theoretical Nanoscience*, 7:1572–1580, 2010.

Jeffrey DiMarco and Michela Taufer. Performance Impact of Dynamic Parallelism on Different Clustering Algorithms. In Eric J. Kelmelis, editor, *Modeling and Simulation for Defense Systems and Applications VIII*, volume 8752, page 87520E. International Society for Optics and Photonics, SPIE, 2013.

Xiangyu Dong, Cong Xu, Yuan Xie, and Norman P. Jouppi. NVSim: A Circuit-Level Performance, Energy, and Area Model for Emerging Nonvolatile Memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(7):994–1007, 2012.

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

Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi Iyer, Dennis Sylvester, David Blaaauw, and Reetuparna Das. Neural Cache: Bit-Serial In-Cache Acceleration of Deep Neural Networks. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 383–396, 2018.

Conal Elliott. Programming Graphics Processors Functionally. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, page 45–56, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138504.

M. Engels, S. Hudson, and C. Magri. GPU-Accelerated Algorithm for Asteroid Shape Modeling. *Astronomy and Computing*, 28:100285, jul 2019. ISSN 22131337.

Jennifer Faj, Tobias Kenter, Sara Faghih-Naini, Christian Plessl, and Vadym Aizinger. Scalable Multi-FPGA Design of a Discontinuous Galerkin Shallow-Water Model on Unstructured Meshes. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '23, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701900.

Kayvon Fatahalian and Mike Houston. GPUs: A Closer Look: As the Line between GPUs and CPUs Begins to Blur, It's Important to Understand What Makes GPUs Tick. *Queue*, 6(2):18–28, mar 2008. ISSN 1542-7730.

Arash Fathi, Loukas F. Kallivokas, and Babak Poursartip. Full-Waveform Inversion in Three-Dimensional PML-Truncated Elastic Media. *Computer Methods in Applied Mechanics and Engineering*, 296:39–72, 2015a. ISSN 0045-7825.

Arash Fathi, Babak Poursartip, and Loukas F. Kallivokas. Time-Domain Hybrid Formulations for Wave Simulations in Three-Dimensional PML-Truncated Heterogeneous Media. *International Journal for Numerical Methods in Engineering*, 101(3):165–198, 2015b.

Arash Fathi, Babak Poursartip, Kenneth H. Stokoe II, and Loukas F. Kallivokas. Three-dimensional P- and S-wave Velocity Profiling of Geotechnical Sites using Full-Waveform Inversion Driven by Field Data. *Soil Dynamics and Earthquake Engineering*, 87:63–81, 2016. ISSN 0267-7261.

Ben Feinberg, Uday Kumar Reddy Vengalam, Nathan Whitehair, Shibo Wang, and Engin Ipek. Enabling Scientific Computing on Memristive Accelerators. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 367–382, 2018.

Ko-An Feng, Chun-Hao Teng, and Min-Hung Chen. A Pseudospectral Penalty Scheme for 2D Isotropic Elastic Wave Computations. *Journal of Scientific Computing*, 33(3):313–348, 2007.

Milinda Fernando, David Neilsen, Hyun Lim, Eric Hirschmann, and Hari Sundar. Massively Parallel Simulations of Binary Black Hole Intermediate-Mass-Ratio Inspirals. *SIAM Journal on Scientific Computing*, 41(2):C97–C138, 2019.

Milinda Fernando, David Neilsen, Eric Hirschmann, Yosef Zlochower, Hari Sundar, Omar Ghattas, and George Biros. A GPU-Accelerated AMR Solver for Gravitational Wave Propagation. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2022.

Milinda Shayamal Fernando and Hari Sundar. paralab/Dendro-5.01: Local timestepping on octree grids , June 2020.

Wilson W.L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 407–420, 2007.

Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett,

Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, Concept, and Design of A Next Generation MPI Implementation. In Dieter Kranzlmüller, Péter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-30218-6.

B. Gallet and M. Gowanlock. Leveraging GPU Tensor Cores for Double Precision Euclidean Distance Calculations. In *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 135–144, Los Alamitos, CA, USA, dec 2022. IEEE Computer Society.

Rajesh Gandham, David Medina, and Timothy Warburton. GPU Accelerated Discontinuous Galerkin Methods for Shallow Water Equations. *Communications in Computational Physics*, 18(1):37–64, 2015. ISSN 19917120.

Congming Gao, Xin Xin, Youyou Lu, Youtao Zhang, Jun Yang, and Jiwu Shu. ParaBit: Processing Parallel Bitwise Operations in NAND Flash Memory Based SSDs. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 59–70, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385572.

Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. ComputeDRAM: In-Memory Compute Using Off-the-Shelf DRAMs. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 100–113, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450369381.

Christophe Geuzaine and Jean-François Remacle. Gmsh: A 3-D Finite Element Mesh Generator with Built-in Pre- and Post-processing Facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331, 2009.

Saugata Ghose, Kevin Hsieh, Amirali Boroumand, Rachata Ausavarungnirun, and Onur Mutlu. *The Processing-in-Memory Paradigm: Mechanisms to Enable Adoption*, chapter 5, pages 133–194. Springer International Publishing, Cham, 2019. ISBN 978-3-319-90385-9.

Dimitrios Gourounas, Bagus Hanindhito, Arash Fathi, Dimitar Trenev, Lizy John, and Andreas Gerstlauer. LAWS: Large-Scale Accelerated Wave Simulations on FPGAs. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '23, page 230, New York, NY, USA, 2023a. Association for Computing Machinery. ISBN 9781450394178.

Dimitrios Gourounas, Bagus Hanindhito, Arash Fathi, Dimitar Trenev, Lizy K. John, and Andreas Gerstlauer. FAWS: FPGA Acceleration of Large-Scale Wave Simulations. In *2023 IEEE 34th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 76–84, 2023b.

DC Groeneveld, LKH Leung, PL Kirillov, VP Bobkov, IP Smogalev, VN Vinogradov, XC Huang, and E Royer. The 1995 Look-Up Table for Critical Heat Flux in Tubes. *Nuclear Engineering and design*, 163(1-2):1–23, 1996.

William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Comput.*, 22(6):789–828, September 1996. ISSN 0167-8191.

Marcus J. Grote, Anna Schneebeli, and Dominik Schötzau. Discontinuous Galerkin Finite Element Method for The Wave Equation. *SIAM Journal on Numerical Analysis*, 44(6):2408–2431, 2006.

Lluís Guasch, Oscar Calderon Agudo, Meng-Xing Tang, Parashkev Nachev, and Michael Warner. Full-Waveform Inversion Imaging of The Human Brain. *npj Digital Medicine*, 3:1 – 12, 2020. ISSN 2398-6352.

Ping Guo and Liqiang Wang. Auto-Tuning CUDA Parameters for Sparse Matrix-Vector Multiplication on GPUs. In *2010 International Conference on Computational and Information Sciences*, pages 1154–1157, 2010.

Kshitij Gupta, Jeff A. Stuart, and John D. Owens. A Study of Persistent Threads style GPU Programming for GPGPU Workloads. In *2012 Innovative Parallel Computing (InPar)*, pages 1–14, 2012.

John L. Gustafson and Isaac T. Yonemoto. Beating Floating Point at its Own Game: Posit Arithmetic. *Supercomputing Frontiers and Innovations*, 4(2): 71–86, Apr. 2017.

Ramyad Hadidi, Lifeng Nai, Hyojong Kim, and Hyesoon Kim. CAIRO: A Compiler-Assisted Technique for Enabling Instruction-Level Offloading of Processing-In-Memory. *ACM Trans. Archit. Code Optim.*, 14(4), December 2017. ISSN 1544-3566.

R. Haelterman, J. Vierendeels, and D. Van Heule. A Generalization of The Runge–Kutta Iteration. *Journal of Computational and Applied Mathematics*, 224(1):152–167, 2009. ISSN 0377-0427.

Ameer Haj-Ali, Rotem Ben-Hur, Nimrod Wald, and Shahar Kvatinsky. Efficient Algorithms for In-Memory Fixed Point Multiplication Using MAGIC. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2018.

Swapan Kumar Haldar. *Mineral Exploration: Principles and Applications*. Elsevier, 2018. ISBN 9780128140222.

Nicholas Hale and Lloyd N. Trefethen. New Quadrature Formulas from Conformal Maps. *SIAM Journal on Numerical Analysis*, 46(2):930–948, 2008.

Khaled Hamidouche, Akshay Venkatesh, Ammar Ahmad Awan, Hari Subramoni, Ching-Hsiang Chu, and Dhabaleswar K. Panda. Exploiting GPUDirect RDMA in Designing High Performance OpenSHMEM for NVIDIA GPU Clusters. In *2015 IEEE International Conference on Cluster Computing*, pages 78–87, 2015.

Tianyi David Han and Tarek S. Abdelrahman. Reducing Branch Divergence in GPU Programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450305693.

Tianyi David Han and Tarek S. Abdelrahman. Reducing Divergence in GPGPU Programs with Loop Merging. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, page 12–23, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320177.

Bagus Hanindhito. GPU-Accelerated High-Performance Computing for Architecture-Aware Wave Simulation Based on Discontinuous Galerkin Algorithms. *UT Electronic Theses and Dissertations*, 2020.

Bagus Hanindhito and Lizy K. John. Accelerating ML Workloads using GPU Tensor Cores: The Good, the Bad, and the Ugly. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*, ICPE '24, page 178–189, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704444.

Bagus Hanindhito, Ruihao Li, Dimitrios Gourounas, Arash Fathi, Karan Govil, Dimitar Trenev, Andreas Gerstlauer, and Lizy John. Wave-PIM: Accelerating Wave Simulation Using Processing-in-Memory. In *Proceedings of the 50th International Conference on Parallel Processing*, ICPP '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450390682.

Bagus Hanindhito, Dimitrios Gourounas, Arash Fathi, Dimitar Trenev, Andreas Gerstlauer, and Lizy K. John. GAPS: GPU-Acceleration of PDE Solvers for Wave Simulation. In *Proceedings of the 36th ACM International Conference on Supercomputing*, ICS '22, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392815.

Bagus Hanindhito, Bhavesh Patel, and Lizy K. John. Bandwidth Characterization of DeepSpeed on Distributed Large Language Model Training. In *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 241–256, 2024.

Mark Harris. Many-Core GPU Computing with NVIDIA CUDA. In *Proceedings of the 22nd Annual International Conference on Supercomputing*, ICS '08, page 1, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605581583.

Alexander Heinecke, Alexander Breuer, and Yifeng Cui. Tensor-Optimized Hardware Accelerates Fused Discontinuous Galerkin Simulations. *Parallel Computing*, 89, 2019. ISSN 01678191.

Moises Hernandez-Fernandez, Istvan Reguly, Saad Jbabdi, Mike Giles, Stephen Smith, and Stamatios N. Sotiropoulos. Using GPUs to Accelerate Computational Diffusion MRI: From Microstructure Estimation to Tractography and Connectomes. *NeuroImage*, 188:598–615, mar 2019. ISSN 10538119.

Jan S. Hesthaven and Tim Warburton. *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications*. Texts in Applied Mathematics. Springer, 2010. ISBN 9781441924636.

Barak Hoffer, Nicolás Wainstein, Christopher M. Neumann, Eric Pop, Eilam Yalon, and Shahar Kvatinsky. Stateful Logic Using Phase Change Memory. *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits*, 8(2):77–83, 2022.

S. B. Hong, N. Vlahopoulos, R. M. Mantey, and D. J. Gorsich. A Computational Approach for Evaluating The Probability of Acoustic Detection of a Military Vehicle. In Wendell R. Watkins, Dieter Clement, and William R. Reynolds, editors, *Targets and Backgrounds X: Characterization and Representation*, volume 5431, pages 150 – 159. International Society for Optics and Photonics, SPIE, 2004.

Masashi Horikoshi, Balazs Gerofi, Yutaka Ishikawa, and Kengo Nakajima. 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*, HPCAsia 2022 Workshop, page 29–39, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450395649.

IBM Corporation. IBM Spectrum MPI: Accelerating High-Performance Application Parallelization, 2024.

Frank Ihlenburg. *Finite Element Analysis of Acoustic Scattering*. Springer-Verlag, 1998. ISBN 0387983198.

Mohsen Imani, Saransh Gupta, and Tajana Rosing. GenPIM: Generalized Processing-in-Memory to Accelerate Data-Intensive Applications. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1155–1158, 2018.

Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. FloatPIM: In-Memory Acceleration of Deep Neural Network Training with High Precision. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 802–815, New York, NY, USA, 2019a. Association for Computing Machinery. ISBN 9781450366694.

Mohsen Imani, Saransh Gupta, Yeseong Kim, Minxuan Zhou, and Tajana Rosing. DigitalPIM: Digital-based Processing-in-Memory for Big Data Acceleration. In *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, GLSVLSI '19, page 429–434, New York, NY, USA, 2019b. Association for Computing Machinery. ISBN 9781450362528.

Mohsen Imani, Saikishan Pampana, Saransh Gupta, Minxuan Zhou, Yeseong Kim, and Tajana Rosing. DUAL: Acceleration of Clustering Algorithms using Digital-based Processing In-Memory. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 356–371, 2020.

Institute of Electrical and Electronics Engineers. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.

Intel Corporation. Intel® MPI Library: Deliver Flexible, Efficient, and Scalable Cluster Messaging., 2024.

Tobin Isaac, Carsten Burstedde, and Omar Ghattas. Low-Cost Parallel Algorithms for 2:1 Octree Balance. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 426–437, 2012.

Tobin Isaac, Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. Recursive Algorithms for Distributed Forests of Octrees. *SIAM Journal on Scientific Computing*, 37(5):C497–C531, 2015.

A. Jameson. The Evolution of Computational Methods in Aerodynamics. *Journal of Applied Mechanics*, 50(4b):1052–1070, 12 1983. ISSN 0021-8936.

H. Jiang. Intel's Ponte Vecchio GPU : Architecture, Systems & Software. In *2022 IEEE Hot Chips 34 Symposium (HCS)*, pages 1–29, Los Alamitos, CA, USA, aug 2022. IEEE Computer Society.

L.F. Kallivokas, A. Fathi, S. Kucukcoban, K.H. Stokoe, J. Bielak, and O. Ghattas. Site Characterization using Full Waveform Inversion. *Soil Dynamics and Earthquake Engineering*, 47:62–82, 2013. ISSN 0267-7261. SI: José Manuel Roësset.

Alex Kanevsky, Mark H. Carpenter, David Gottlieb, and Jan S. Hesthaven. Application of Implicit–Explicit High Order Runge–Kutta Methods to Discontinuous-Galerkin Schemes. *Journal of Computational Physics*, 225(2):1753–1781, 2007. ISSN 0021-9991.

A. Karakus, T. Warburton, M. H. Aksel, and C. Sert. A GPU Accelerated Level Set Reinitialization for An Adaptive Discontinuous Galerkin Method. *Computers and Mathematics with Applications*, 72(3):755–767, 2016. ISSN 08981221.

A. Karakus, N. Chalmers, K. Świrydowicz, and T. Warburton. A GPU Accelerated Discontinuous Galerkin Incompressible Flow Solver. *Journal of Computational Physics*, 390:380–404, 2019. ISSN 10902716.

A.A. Kaufman and A.L. Levshin. *Acoustic and Elastic Wave Fields in Geophysics III*. Elsevier, 2005.

Stephen W. Keckler, William J. Dally, Brucek Khailany, Michael Garland, and David Glasco. GPUs and The Future of Parallel Computing. *IEEE Micro*, 31 (5):7–17, 2011.

Joseph P. Kenny and Craig D. Ulmer. RoCE: Promising Technology for Ethernet as a High Performance Networking Fabric. Technical report, Sandia National Lab. (SNL-CA), United States, 2019.

Gokcen Kestor, Roberto Gioiosa, Darren J. Kerbyson, and Adolfy Hoisie. Quantifying The Energy Cost of Data Movement in Scientific Applications. In *2013*

IEEE International Symposium on Workload Characterization (IISWC), pages 56–65, 2013.

David I. Ketcheson. Runge–Kutta Methods with Minimum Storage Implementations. *Journal of Computational Physics*, 229(5):1763–1773, 2010. ISSN 0021-9991.

Soroosh Khoram, Yue Zha, Jialiang Zhang, and Jing Li. Challenges and Opportunities: From Near-Memory Computing to In-Memory Computing. In *Proceedings of the 2017 ACM on International Symposium on Physical Design*, ISPD '17, page 43–46, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450346962.

Suhwan Kim, Changue Jung, and Younghoon Kim. Comparative Analysis of GPU Stream Processing Between Persistent and Non-Persistent Kernels. In *2022 13th International Conference on Information and Communication Technology Convergence (ICTC)*, pages 2330–2332, 2022.

D Kosloff and H Tal-Ezer. A Modified Chebyshev Pseudospectral Method with an O(N-1) Time Step Restriction. *Journal of Computational Physics*, 104(2): 457–469, 1993. ISSN 0021-9991.

J.E. Kozdon and L.C. Wilcox. An Energy Stable Approach for Discretizing Hyperbolic Equations with Nonconforming Discontinuous Galerkin Methods. *Journal of Scientific Computing*, 76:1742–1784, 2018. ISSN 1573-7691.

Shahar Kvatinsky, Dmitry Belousov, Slavik Liman, Guy Satat, Nimrod Wald, Eby G. Friedman, Avinoam Kolodny, and Uri C. Weiser. MAGIC—Memristor-Aided Logic. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 61(11):895–899, 2014.

Shahar Kvatinsky, Misbah Ramadan, Eby G. Friedman, and Avinoam Kolodny. VTEAM: A General Model for Voltage-Controlled Memristors. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 62(8):786–790, 2015.

Martin-D. Lacasse, Laurent White, Huseyin Denli, and Lingyun Qiu. *Full-Wavefield Inversion: An Extreme-Scale PDE-Constrained Optimization Problem*, pages 205–255. Springer New York, New York, NY, 2018. ISBN 978-1-4939-8636-1.

Ignacio Laguna, Ryan Marshall, Kathryn Mohror, Martin Ruefenacht, Anthony Skjellum, and Nawrin Sultana. 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*, SC '19, pages 1–14, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362290.

Michael Larabel. Intel Launches 4th Gen Xeon Scalable "Sapphire Rapids", Xeon CPU Max Series, 2023.

Wai-Kong Lee, Hwajeong Seo, Zhenfei Zhang, and Seong Oun Hwang. Tensor-Crypto: High Throughput Accel eration of Lattice-Based Cryptography using Tensor Core on GPU. *IEEE Access*, 10:20616–20632, 2022.

Charles E Leiserson. Area-Efficient Graph Layouts. In *21st Annual Symposium on Foundations of Computer Science (sfcs 1980)*, pages 270–281. IEEE, 1980.

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

Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. Evaluating Modern GPU Interconnect: PCIe,

NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110, 2020a.

Feng Li, Yunming Ye, Zhaoyang Tian, and Xiaofeng Zhang. CPU versus GPU: Which Can Perform Matrix Computation Faster—Performance Comparison for Basic Linear Algebra Subprograms. *Neural Computing and Applications*, 31(8): 4353–4365, January 2018b.

Ruihao Li, Shuang Song, Qinzhe Wu, and Lizy K. John. Accelerating Force-Directed Graph Layout with Processing-in-Memory Architecture. In *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 271–282, 2020b.

Shang Li, Dhiraj Reddy, and Bruce Jacob. A performance & Power Comparison of Modern High-Speed DRAM Architectures. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '18, page 341–353, New York, NY, USA, 2018c. Association for Computing Machinery. ISBN 9781450364751.

Yuke Li, Arjun Kashyap, Yanfei Guo, and Xiaoyi Lu. Characterizing Lossy and Lossless Compression on Emerging BlueField DPU Architectures. In *2023 IEEE Symposium on High-Performance Interconnects (HOTI)*, pages 33–40, 2023.

Yuke Li, Arjun Kashyap, Weicong Chen, Yanfei Guo, and Xiaoyi Lu. Accelerating Lossy and Lossless Compression on Emerging BlueField DPU Architectures. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 373–385, 2024.

Robert Lim, Boyana Norris, and Allen Malony. Autotuning GPU Kernels via Static and Predictive Analysis. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 523–532, 2017.

Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2): 39–55, 2008.

Peter Lindstrom and Martin Isenburg. Fast and Efficient Compression of Floating-Point Data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1245–1250, 2006.

Elena Lucano, Micaela Liberti, Gonzalo G. Mendoza, Tom Lloyd, Maria Ida Iacono, Francesca Apollonio, Steve Wedan, Wolfgang Kainz, and Leonardo M. Angelone. Assessing the Electromagnetic Fields Generated by a Radiofrequency MRI Body Coil at 64 MHz: Defeaturing Versus Accuracy. *IEEE Transactions on Biomedical Engineering*, 63(8):1591–1601, 2016.

Mark E Lucente. Interactive Computation of Holograms using a Look-Up Table. *Journal of Electronic Imaging*, 2(1):28–35, 1993.

Siyuan Ma, Kaustubh Mhatre, Jian Weng, Bagus Hanindhito, Zhengrong Wang, Tony Nowatzki, Lizy John, and Aman Arora. PIMSAB: A Processing-In-Memory System with Spatially-Aware Communication and Bit-Serial-Aware Computation. *ACM Trans. Archit. Code Optim.*, September 2024. ISSN 1544-3566. Just Accepted.

Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical report, University of Tennessee, USA, 1994.

Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*.

Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*.

Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 1.3*.

Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 2.1*.

Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 2.2*.

Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 3.0*.

Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 3.1*.

Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*.

Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.1*.

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed Precision Training. In *International Conference on Learning Representations*, Vancouver, BC, Canada, 2018. Open Review.

Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting Network Support for RDMA. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 313–326, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355674.

A. Modave, A. St-Cyr, and T. Warburton. GPU Performance Analysis of A Nodal Discontinuous Galerkin Method for Acoustic and Elastic Models. *Computers and Geosciences*, 91:64–76, 2016. ISSN 00983004.

Nazmul Haque Mondol. *Seismic Exploration*, pages 375–402. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-02332-3.

Nazmul Haque Mondol. Seismic Exploration. In *Petroleum Geoscience*, volume 41, pages 427–454. Springer Berlin Heidelberg, Berlin, Heidelberg, apr 2015. ISBN 9783642341311.

Dawei Mu, Po Chen, and Liqiang Wang. Accelerating The Discontinuous Galerkin Method for Seismic Wave Propagation Simulations using The Graphic Processing Unit (GPU)-Single-GPU Implementation. *Computers and Geosciences*, 51:282–292, 2013. ISSN 00983004.

Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. Enabling Practical Processing In and Near Memory for Data-Intensive Computing. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, pages 1–4, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367257.

John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable Parallel Programming with CUDA: Is CUDA the Parallel Programming Model That Application Developers Have Been Waiting For? *Queue*, 6(2):40–53, mar 2008a. ISSN 1542-7730.

John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable Parallel Programming with CUDA. In *ACM SIGGRAPH 2008 Classes*, SIGGRAPH '08, New York, NY, USA, 2008b. Association for Computing Machinery. ISBN 9781450378451.

Jens Niegemann, Richard Diehl, and Kurt Busch. Efficient Low-Storage Runge–Kutta Schemes with Optimized Stability Regions. *Journal of Computational Physics*, 231(2):364–372, 2012. ISSN 0021-9991.

C.F. Nielsen. GPU Accelerated Simulation of Channeling Radiation of Relativistic Particles. *Computer Physics Communications*, page 107128, dec 2019. ISSN 00104655.

Frank Nielsen. *Introduction to MPI: The Message Passing Interface*, pages 21–62. Springer International Publishing, Cham, 2016. ISBN 978-3-319-21903-5.

Ritesh Nohria, Gustavo Santos, and Volker Haug. IBM Power System AC922: Technical Overview and Introduction. Redpaper, International Business Machines Corporation, New York, US, July 2018.

NVIDIA Corporation. NVIDIA Tesla P100: The Most Advanced Datacenter Accelerator Ever Built Featuring Pascal GP100, The World's Fastest GPU. Whitepaper, NVIDIA Corporation, US, January 2016.

NVIDIA Corporation. NVIDIA Tesla V100 GPU Architecture: The World's Most Advanced Data Center GPU. Whitepaper, NVIDIA Corporation, California, US, August 2017.

NVIDIA Corporation. NVIDIA Turing GPU Architecture: Graphics Reinvented. Whitepaper, NVIDIA Corporation, California, US, September 2018.

NVIDIA Corporation. NVIDIA A100 Tensor Core GPU Architecture: Unprecedented Acceleration at Every Scale. Whitepaper, NVIDIA Corporation, California, US, January 2020a.

NVIDIA Corporation. NVIDIA DGX A100: The Universal System for AI Infrastructure. Datasheet, NVIDIA Corporation, US, January 2020b.

NVIDIA Corporation. NVIDIA System Management Interface, 2020c.

NVIDIA Corporation. NVIDIA H100 Tensor Core GPU Architecture: Exceptional Performance, Scalability, and Security for The Data Center. Whitepaper, NVIDIA Corporation, California, US, January 2022.

NVIDIA Corporation. Nsight Compute CLI, 2023a.

NVIDIA Corporation. NVIDIA Ada GPU Architecture: Designed to Deliver Outstanding Gaming and Creating, Professional Graphics, AI, and Compute Performance. Whitepaper, NVIDIA Corporation, California, US, January 2023b.

NVIDIA Corporation. Profiler User's Guide, 2024.

E. Oktay, N. Alemdaroglu, E. Tarhan, P. Champigny, and P. d'Espiney. Euler and Navier-Stokes Solutions for Missiles at High Angles of Attack. *Journal of Spacecraft and Rockets*, 36(6):850–858, 1999.

Dhabaleswar Kumar Panda, Hari Subramoni, Ching-Hsiang Chu, and Mohammadreza Bayatpour. The MVAPICH project: Transforming Research Into High-Performance MPI Library for HPC Community. *Journal of Computational Science*, 52:101208, 2021. ISSN 1877-7503. Case Studies in Translational Computer Science.

paralab. Dendro-gr: Numerical relativity with octree based wavelet adaptive mesh refinement, 2021.

Seymour V. Parter. On The Legendre–Gauss–Lobatto Points and Weights. *J. Sci. Comput.*, 14(4):347–355, dec 1999. ISSN 0885-7474.

Hagen Peters, Martin Köper, and Norbert Luttenberger. Efficiently Using a CUDA-enabled GPU as Shared Resource. In *2010 10th IEEE International Conference on Computer and Information Technology*, pages 1122–1127, 2010.

Tomasz Plewa, Timur Linde, and V. Gregory Weirs, editors. *Adaptive Mesh Refinement - Theory and Applications: Proceedings of the Chicago Workshop on Adaptive Mesh Refinement Methods, Sept. 3–5, 2003*. Springer Berlin Heidelberg, 2005. ISBN 9783540270393.

Sreeram Potluri, Khaled Hamidouche, Akshay Venkatesh, Devendar Bureddy, and Dhabaleswar K. Panda. Efficient Inter-node MPI Communication Using GPUDirect RDMA for InfiniBand Clusters with NVIDIA GPUs. In *2013 42nd International Conference on Parallel Processing*, pages 80–89, 2013.

Babak Poursartip, Arash Fathi, and Loukas F. Kallivokas. Seismic Wave Amplification by Topographic Features: A Parametric Study. *Soil Dynamics and Earthquake Engineering*, 92:503–527, 2017. ISSN 0267-7261.

Babak Poursartip, Arash Fathi, and John L. Tassoulas. Large-scale Simulation of Seismic Wave Motion: A Review. *Soil Dynamics and Earthquake Engineering*, 129:105909, 2020. ISSN 0267-7261.

A. Quarteroni and A. Valli. *Numerical Approximation of Partial Differential Equations*. Springer Series in Computational Mathematics. Springer, 1994. ISBN 978-3-540-85267-4.

Raspberry Pi Foundation. Raspberry pi power supply documentation, 2021.

P. Ratanaworabhan, Jian Ke, and M. Burtscher. Fast Lossless Compression of Scientific Floating-Point Data. In *Data Compression Conference (DCC'06)*, pages 133–142, 2006.

Prashant Singh Rawat, Fabrice Rastello, Aravind Sukumaran-Rajam, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. Register Optimizations for Stencils on GPUs. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, page 168–182, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450349826.

J.-F. Remacle, R. Gandham, and T. Warburton. GPU Accelerated Spectral Finite Elements on All-Hex Meshes. *Journal of Computational Physics*, 324: 246–257, nov 2016. ISSN 00219991.

Feng Ren, Baowei Song, Ya Zhang, and Haibao Hu. A GPU-Accelerated Solver for Turbulent Flow and Scalar Transport Based on The Lattice Boltzmann Method. *Computers & Fluids*, 173:29–36, sep 2018. ISSN 00457930.

K. Sankaran. Recent Trends in Computational Electromagnetics for Defence Applications. *Defence Sience Journal*, 69:65–73, 2019.

Katsuto Sato, Hiroyuki Takizawa, Kazuhiko Komatsu, and Hiroaki Kobayashi. *Automatic Tuning of CUDA Execution Parameters for Stencil Processing*, pages 209–228. Springer New York, New York, NY, 2010. ISBN 978-1-4419-6935-4.

David Schneider. The Exascale Era is Upon Us: The Frontier supercomputer may be the first to reach 1,000,000,000,000,000,000 operations per second. *IEEE Spectrum*, 59(1):34–35, 2022.

Richard Arnoud Schoonhoven, Ben van Werkhoven, and Kees Joost Batenburg. Benchmarking Optimization Algorithms for Auto-Tuning GPU Kernels. *IEEE Transactions on Evolutionary Computation*, 27(3):550–564, 2023.

R. L. Sengbush. *Seismic Exploration Methods*. IHRDC Boston, Boston, 1983. ISBN 0934634211; 9780934634212; 9789401163996; 9401163995; 9789401163972; 9401163979.

Bruno Seny, Jonathan Lambrechts, Thomas Toulorge, Vincent Legat, and Jean-François Remacle. An Efficient Parallel Implementation of Explicit Multirate Runge–Kutta Schemes for Discontinuous Galerkin Computations. *Journal of Computational Physics*, 256:135–160, 2014. ISSN 0021-9991.

Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 273–287,

New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349529.

Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 14–26, 2016.

S. S. Sharkawi and G. A. Chochia. Communication Protocol Optimization for Enhanced GPU Performance. *IBM Journal of Research and Development*, 64 (3/4):9:1–9:9, 2020.

Yongwon Shin, Juseong Park, Sungjun Cho, and Hyojin Sung. PIMFlow: Compiler and Runtime Support for CNN Models on Processing-in-Memory DRAM. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, CGO '23, page 249–262, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701016.

Shuli Shu, Jingchang Zhang, and Ning Yang. GPU-Accelerated Transient Lattice Boltzmann Simulation of Bubble Column Reactors. *Chemical Engineering Science*, 214:115436, mar 2020. ISSN 00092509.

Anne Siemon, Stephan Menzel, Rainer Waser, and Eike Linn. A Complementary Resistive Switch-Based Crossbar Array Adder. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 5(1):64–74, 2015.

Simo-Pekka Simonaho, Timo Lähivaara, and Tomi Huttunen. Modeling of Acoustic Wave Propagation in Time-Domain using The Discontinuous Galerkin Method – A Comparison with Measurements. *Applied Acoustics*, 73(2):173–183, 2012. ISSN 0003-682X.

Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 541–552, 2017.

Aleksei Sorokin, Sergey Malkovsky, and Georgiy Tsoy. Comparing The Performance of General Matrix Multiplication Routine on Heterogeneous Computing Systems. *Journal of Parallel and Distributed Computing*, 160:39–48, 2022. ISSN 0743-7315.

Dan. Stanzione. Thoughts from TACC on next generation academic supercomputing systems, 2022.

Sayantan Sur, Matthew J. Koop, and Dhabaleswar K. Panda. High-Performance and Scalable MPI Over InfiniBand with Reduced Memory Usage: An In-Depth Performance Analysis. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, page 105–es, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 0769527000.

Nishil Talati, Saransh Gupta, Pravin Mane, and Shahar Kvatinsky. Logic Design Within Memristive Memories Using Memristor-Aided loGIC (MAGIC). *IEEE Transactions on Nanotechnology*, 15(4):635–650, 2016.

Xulong Tang, Ashutosh Pattnaik, Huaipan Jiang, Onur Kayiran, Adwait Jog, Sreepathi Pai, Mohamed Ibrahim, Mahmut T. Kandemir, and Chita R. Das. Controlled Kernel Launch for Dynamic Parallelism in GPUs. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 649–660, 2017.

TechPowerUp. NVIDIA Tesla P100 PCIe 16 GB, 2016.

TechPowerUp. NVIDIA GeForce GTX 1080 Ti, 2017a.

TechPowerUp. NVIDIA Tesla V100 PCIe 16 GB, 2017b.

TechPowerUp. NVIDIA Tesla V100 SXM2 16 GB, 2017c.

TechPowerUp. NVIDIA A100 PCIe 40 GB, 2020.

A. Tekin, A.Tuncer Durak, C. Piechurski, D. Kaliszan, F. Aylin Sungur, F. Robertsén, and P. Gschwandtner. State-of-the-Art and Trends for Computing and Interconnect Network Solutions for HPC and AI. *Partnership for Advanced Computing in Europe*, 2021.

Texas Advanced Computing Center. TACC Longhorn User Guide, 2020a.

Texas Advanced Computing Center. Maverick2 User Guide, 2020b.

Texas Advanced Computing Center. Lonestar6 User Guide, 2024.

Carsten Uphoff and Michael Bader. Yet Another Tensor Toolbox for Discontinuous Galerkin Methods and Other Applications. *ACM Trans. Math. Softw.*, 46(4), October 2020. ISSN 0098-3500.

P.J. van der Houwen. The Development of Runge-Kutta Methods for Partial Differential Equations. *Applied Numerical Mathematics*, 20(3):261–272, 1996. ISSN 0168-9274.

G.A.E. Vandenbosch. Computational Electromagnetics and Antenna Design in Western Europe Organisation and Overview of The Present Status. In *The Fifth International Kharkov Symposium on Physics and Engineering of Microwaves, Millimeter, and Submillimeter Waves (IEEE Cat. No.04EX828)*, volume 1, pages 40–45 Vol.1, 2004.

A. Venkatesh, H. Subramoni, K. Hamidouche, and Dhabaleswar K. Panda. A High Performance Broadcast Design with Hardware Multicast and GPUDirect

RDMA for Streaming Applications on Infiniband Clusters. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10, 2014.

Lucas Vespa, Alexander Bauman, and Jenny Wells. Algorithm Flattening: Complete Branch Elimination for GPU Requires A Paradigm Shift from CPU Thinking. In *2015 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2015.

Oriol Vinyals and Gerald Friedland. A Hardware-Independent Fast Logarithm Approximation with Adjustable Accuracy. In *2008 Tenth IEEE International Symposium on Multimedia*, pages 61–65, 2008.

Mohamed Wahib and Naoya Maruyama. Scalable Kernel Fusion for Memory-Bound GPU Applications. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 191–202, 2014.

Wubing Wan, Lin Gan, Wenqiang Wang, Zekun Yin, Haodong Tian, Zhenguo Zhang, Yinuo Wang, Mengyuan Hua, Xiaohui Liu, Shengye Xiang, Zhongqiu He, Zijia Wang, Ping Gao, Xiaohui Duan, Weiguo Liu, Wei Xue, Haohuan Fu, Guangwen Yang, Xiaofei Chen, Zeyu Song, Yaojian Chen, Xin Liu, and Wei Zhang. 69.7-PFlops Extreme Scale Earthquake Simulation with Crossing Multi-faults and Topography on Sunway. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701092.

Jiajun Wang, Ahmed Khawaja, George Biros, Andreas Gerstlauer, and Lizy K. John. Optimizing GPGPU Kernel Summation for Performance and Energy Efficiency. In *2016 45th International Conference on Parallel Processing Workshops (ICPPW)*, pages 123–132, 2016a.

Jin Wang and Sudhakar Yalamanchili. Characterization and Analysis of Dynamic Parallelism in Unstructured GPU Applications. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 51–60, 2014.

Jin Wang, Norm Rubin, Albert Sidelnik, and Sudhakar Yalamanchili. LaPerm: Locality Aware Scheduler for Dynamic Parallelism on GPUs. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, page 583–595. IEEE Press, 2016b. ISBN 9781467389471.

Qian Wang, Tianyu Wang, Zhaoyan Shen, Zhiping Jia, Mengying Zhao, and Zili Shao. Re-tangle: A reram-based processing-in-memory architecture for transaction-based blockchain. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2019a.

Qian Wang, Zhiping Jia, Tianyu Wang, Zhaoyan Shen, Mengying Zhao, Renhai Chen, and Zili Shao. A Highly Parallelized PIM-Based Accelerator for Transaction-Based Blockchain in IoT Environment. *IEEE Internet of Things Journal*, 7(5):4072–4083, 2020.

Shibo Wang and Pankaj Kanwar. BFloat16: The Secret to High Performance on Cloud TPUs. *Google Cloud Blog*, 4, 2019.

Yahui Wang, Yu Ma, and Ming Xie. GPU Accelerated Lattice Boltzmann Method in Neutron Kinetics Problems II: Neutron Transport Calculation. *Annals of Nuclear Energy*, 134:305–317, dec 2019b. ISSN 03064549.

Yahui Wang, Yu Ma, and Ming Xie. GPU Accelerated Lattice Boltzmann Method in Neutron Kinetics Problems. *Annals of Nuclear Energy*, 129:350–365, jul 2019c. ISSN 03064549.

J. Wei and F.E. Kruis. GPU-Accelerated Monte Carlo Simulation of Particle Coagulation Based on The Inverse Method. *Journal of Computational Physics*, 249:67–79, sep 2013. ISSN 00219991.

L.C. Wilcox, G. Stadler, C. Burstedde, and O. Ghattas. A High-Order Discontinuous Galerkin Method for Wave Propagation Through Coupled Elastic–Acoustic Media. *Journal of Computational Physics*, 229(24):9373 – 9396, 2010. ISSN 0021–9991.

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM*, 52(4):65–76, apr 2009. ISSN 0001-0782.

J.H Williamson. Low-storage runge-kutta schemes. *Journal of Computational Physics*, 35(1):48–56, 1980. ISSN 0021-9991.

Greg von Winckel. Legendre-Gauss Quadrature Weights and Nodes, 2024.

Sebastian Wolf, Martin Galis, Carsten Uphoff, Alice-Agnes Gabriel, Peter Moczo, David Gregor, and Michael Bader. An efficient ADER-DG local time stepping scheme for 3D HPC simulation of seismic waves in poroelastic media. *Journal of Computational Physics*, 455:110886, 2022. ISSN 0021-9991.

Bo Wu, Zhijia Zhao, Eddy Zheng Zhang, Yunlian Jiang, and Xipeng Shen. 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*, PPoPP '13, page 57–68, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450319225.

Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. Enabling and Exploiting Flexible Task Assignment on GPU through SM-Centric Program Transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, page 119–130, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450335591.

Wm. A. Wulf and Sally A. McKee. Hitting The Memory Wall: Implications of The Obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995. ISSN 0163-5964.

A. Xu, L. Shi, and T.S. Zhao. Accelerated Lattice Boltzmann Simulation using GPU and OpenACC with Data Management. *International Journal of Heat and Mass Transfer*, 109:577–588, jun 2017. ISSN 00179310.

Charlene Yang. Berkeley CS Roofline Toolkit, 2015.

Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, and Xipeng Shen. Streamlining GPU Applications on The Fly: Thread Divergence Elimination Through Runtime Thread-Data Remapping. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, page 115–126, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300186.

Peter Zhang, Eric Holk, John Matty, Samantha Misurda, Marcin Zalewski, Jonathan Chu, Scott McMillan, and Andrew Lumsdaine. Dynamic Parallelism for Simple and Efficient GPU Graph Algorithms. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, IA¡sup¿3¡/sup¿ '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450340014.

Xingyao Zhang, Shuaiwen Leon Song, Chenhao Xie, Jing Wang, Weigong Zhang, and Xin Fu. Enabling highly efficient capsule networks processing through a pim-based architecture design. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 542–555, 2020a.

Yuhao Zhang, Zhiping Jia, Yungang Pan, Hongchao Du, Zhaoyan Shen, Mengying Zhao, and Zili Shao. PattPIM: A Practical ReRAM-Based DNN Accelerator by Reusing Weight Pattern Repetitions. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020b.

Han Zhao, Weihao Cui, Quan Chen, Jieru Zhao, Jingwen Leng, and Minyi Guo. Exploiting Intra-SM Parallelism in GPUs via Persistent and Elastic Blocks. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pages 290–298, 2021.

Xu-dong Zhao, Shu-xiu Liang, Zhao-chen Sun, Xi-zeng Zhao, Jia-wen Sun, and Zhong-bo Liu. A GPU Accelerated Finite Volume Coastal Ocean Model. *Journal of Hydrodynamics*, 29(4):679–690, aug 2017. ISSN 1001-6058.

Scott Zuloaga, Rui Liu, Pai-Yu Chen, and Shimeng Yu. Scaling 2-Layer RRAM Cross-Point Array Towards 10 nm Node: A Device-Circuit Co-Design. In *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 193–196, 2015.

Vita

Bagus Hanindhito was born in Yogyakarta, Indonesia. He completed his Bachelor of Science (B.S.) in Electrical Engineering at Institut Teknologi Bandung, Indonesia, in 2015. He then worked as an integrated circuit physical design engineer at a fabless semiconductor company for almost two years. He returned to school in 2017 and received his Master of Science (M.S.) degree in Electrical Engineering from Institut Teknologi Bandung, Indonesia, in 2019. His research was on hardware security, particularly developing methods to detect hardware trojans using side-channel analysis. After observing the stagnancy of computing performance over the last decades due to power, complexity, and memory walls, he wants to take part in research on future computer systems. However, the industry and research in these fields are not available in Indonesia, and thus, he pursued his dream abroad. In 2018, he was admitted as a master's student at the Electrical and Computer Engineering Department at the University of Texas at Austin. He received Master of Science in Engineering (M.S.E.) degree in 2020. Under the same research team, he continued his research as a Ph.D. student interested in GPU-accelerated high-performance computing, workload characterization, performance evaluation and optimizations, and emerging computing technologies. In December 2024, he earned his Ph.D. and planned to continue his research in his field of interest by taking a research position in the industry.

Address: hanindhito@bagus.my.id

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.