Performance Modeling, Optimization, and Characterization on Heterogeneous Architectures

Lokendra Singh Panwar

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Wu-chun Feng, Chair
Yong Cao
Peter M. Athanas

January 31, 2014
Blacksburg, Virginia

Keywords: Heterogeneous Computing, Graphics Processing Unit (GPU), GPU Emulation, Performance Modeling, Finite Difference Method, Seismology Modeling

Performance Modeling, Optimization, and Characterization on Heterogeneous Architectures

Lokendra Singh Panwar

(ABSTRACT)

Today, heterogeneous computing has truly reshaped the way scientists think and approach high-performance computing (HPC). Hardware accelerators such as general-purpose graphics processing units (GPUs) and Intel Many Integrated Core (MIC) architecture continue to make in-roads in accelerating large-scale scientific applications. These advancements, however, introduce new sets of challenges to the scientific community such as: selection of best processor for an application, effective performance optimization strategies, maintaining performance portability across architectures etc. In this thesis, we present our techniques and approach to address some of these significant issues.

Firstly, we present a fully automated approach to project the relative performance of an OpenCL program over different GPUs. Performance projections can be made within a small amount of time, and the projection overhead stays relatively constant with the input data size. As a result, the technique can help runtime tools make dynamic decisions about which GPU would run faster for a given kernel. Usage cases of this technique include scheduling or migrating GPU workloads over a heterogeneous cluster with different types of GPUs.

We then present our approach to accelerate a seismology modeling application that is based on the finite difference method (FDM), using MPI and CUDA over a hybrid CPU+GPU cluster. We describe the generic computational complexities involved in porting such applications to the GPUs and present our strategy of efficient performance optimization and characterization. We also show how performance modeling can be used to reason and drive the hardware-specific optimizations on the GPU. The performance evaluation of our approach delivers a maximum speedup of 23-fold with a single GPU and 33-fold with dual GPUs per node over the serial version of the application, which in turn results in a many-fold speedup when coupled with the MPI distribution of the computation across the cluster. We also study the efficacy of GPU-integrated MPI, with MPI-ACC as an example implementation, on a seismology modeling application and discuss the lessons learned.

This work was supported in part by the Department of Energy contract number DE-EE0002758 via the American Recovery and Reinvestment Act (ARRA) of 2009.

Dedication

I dedicate this thesis to my mom, dad, and sister.

Acknowledgments

I would like to express my heartfelt gratitude to my advisor Dr. Wu-chun Feng, foremost, for his invaluable guidance and support throughout my M.S. studies. His belief in his students and encouragement to think independently to develop solutions to the research problems made the overall research experience greatly rewarding. His diligent work-ethic and dynamic personality had been a wonderful source of inspiration to me.

It has been a pleasure working with Dr. Pavan Balaji and Dr. Jiayuan Meng from Argonne National Lab, and I thank them for all their help and advise during our two productive collaborative projects.

I am very thankful to Dr. Yong Cao and Dr. Peter Athanas for agreeing to serve on my M.S. committee and for their valuable comments and suggestions to improve this thesis.

I had a great time working with the SyNeRGy team. Special thanks to Ashwin Aji, for being an amazing friend and colleague at the lab. As a newcomer to research, I learnt a lot from his earnest approach towards research and our brainstorming sessions. Many thanks to Tom, for all his technical expertise and help; Balaji and Umar, for discussions and their advises related to *grad life*; Nabeel, Sriram and Vignesh for being wonderful friends and for some fun lunch/dinner time discussions; Carlo, Heshan, Konstantinos, Hao, Jing, Kaixi, Mariam, Paul and Dr. Trease for their constructive comments and help with my research.

I would also like to thank my other friends at Virginia Tech, particularly Naresh, Prasad, Sushil, Parang, Ritesh, Ankam, Sam, Amiya, Arka, Bharath and Apoorva for making these two years memorable.

Last, but not the least, I dedicate this thesis to my parents and my sister for their unconditional love, care and encouragement all my life. They have been immensely supportive of all my academic and professional endeavors and have made me what I am today.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In the past decade, the landscape of high-performance computing (HPC) has seen itself evolving towards heterogeneous parallel computing, with hardware accelerators being used as co-processors to accelerate parallel components of scientific applications. Recently released Top500 list in November 2013, that ranks world's most powerful 500 supercomputers, indicates that 53 systems use accelerators as co-processors; up from four in 2008 (Figure 1.1).

The success of heterogeneous computing is primarily attributed to high throughput computations at low cost and better energy efficiency; effectively measured by metrics such as FLOPS/dollar and FLOPS/watt (FLOPS: floating point operations per second). For example, in the November 2013 Top500 list, while only 10.3% of the top 500 supercomputers used

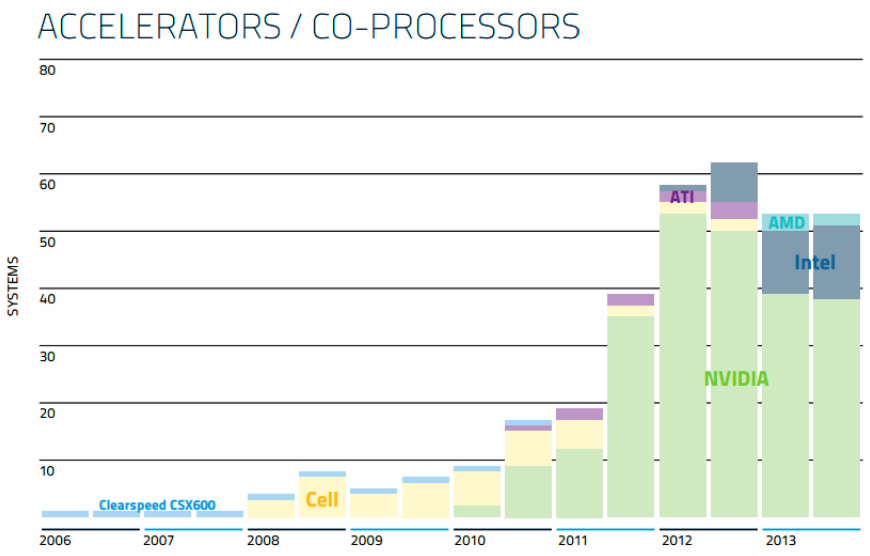
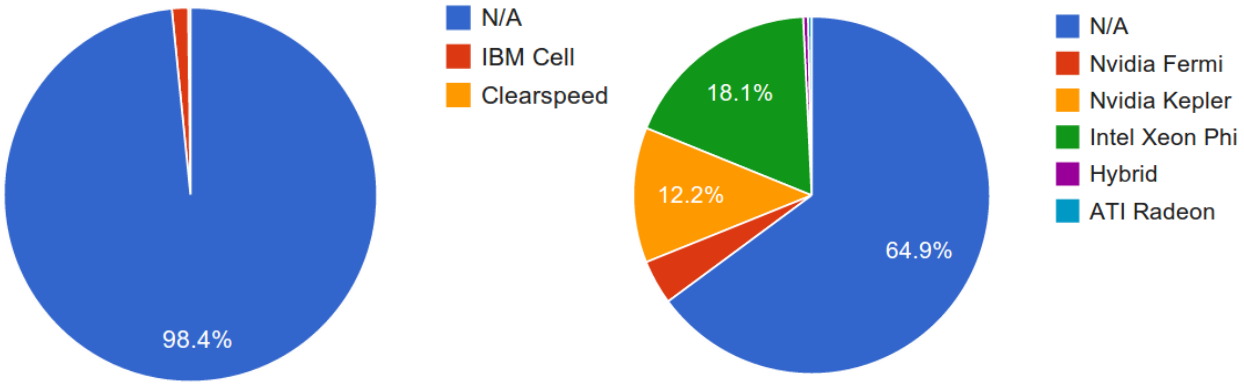


Figure 1.1: Top500 Supercomputers Deploying Accelerators/Co-Processors. [*TOP500 list, November 2013. http://www.top500.org/lists/2013/11. Used under fair use, 2014*]

accelerators, their share in the overall performance was 33.7%; which primarily is the result of massive parallel arithmetic computation capabilities of modern accelerators such as GPUs and Intel MIC accelerators.

Another interesting observable trend in the Top500 lists of past few years has been the increasing diversity in the families and types of accelerators being deployed (Figure 1.2b). For example, while different accelerators such as GPUs and Intel Xeon Phi co-processors power modern supercomputers, there exists substantial diversity within the GPUs, with multiple architectural families of GPUs provided by different vendors such as NVIDIA and AMD. In particular, GPUs have gained widespread use as general-purpose computational accelerators and have been studied extensively across a broad range of scientific applications [16, 21, 34, 43]. In fact, of the 53 systems in the top 500 supercomputers that deploy general-purpose accelerators, 40 use GPUs [1].



(a) 2008 (b) 2013

Figure 1.2: Performance Share of Accelerators in Top500 List. [*TOP500 list, November 2013. http://www.top500.org/lists/2013/11. Used under fair use, 2014*]

While such computational horsepower at low cost and high energy efficiency appears to be a 'win-win' situation, heterogeneous computing presents its own set of challenges that requires a new, and arguably unconventional, approach to address these new issues. The primary challenges are as follows:

1. Programming Models: Different programming models supported by the accelerators lead to added burden on the programmer to create several versions of the application. While projects such as OpenCL [22] have been initiated to establish a standard programming model, it is a low-level language that cannot preserve performance portability across platforms.

2. Best Device for an Application: The accelerators often specialize in accelerating only a certain type of applications; and in the real world, a domain scientist often does not know in advance whether a given accelerator would benefit his/her application, without actually re-writing the application itself and executing it on multiple accelerators.

3. Performance Optimizations and Characterization: A generic heterogeneous application needs to be optimized differently for different accelerators, for their unique architectural properties; which, otherwise, can lead to inefficient utilization of a given accelerator.

1.2 Contributions

This thesis makes the following contributions:

- An online kernel characterization technique that retrieves key performance characteristics with relatively low overhead by emulating the execution of a downscaled kernel.
- An online performance model that projects the performance of full kernel execution from statistics collected from the downscaled emulation.
- An end-to-end implementation of our technique that covers multiple architectural families from both AMD and NVIDIA GPUs.
- Efficient strategies to map a FDM-based seismology application (FDM-Seismology) onto a CPU-GPU heterogeneous cluster.
- Performance characterization using performance models to explain and help drive application-specific optimizations in FDM-Seismology.
- An in-depth analysis of the efficacy of MPI-ACC, a GPU-integrated MPI library, for inter-node data transfer optimizations in FDM-Seismology.

Chapter 2

Background

In this chapter, we briefly describe GPU architectures from AMD and NVIDIA and their associated programming models, including CUDA (Compute Unified Device Architecture), OpenCL (Open Computing Language), VOCL (Virtual OpenCL), MPI (Message Passing Interface), and a GPU-integrated MPI dubbed MPI-ACC, short for MPI for Accelerators. We conclude the chapter with a discussion of seismology modeling based on the finite-difference method (FDM) and stencil computations.

2.1 GPU Architectures

GPUs were primarily designed to accelerate the rendering of images in a frame buffer to be displayed. However, with key architectural changes to the graphics pipeline, GPUs have emerged as massive parallel processing accelerators that can be used to accelerate general-

purpose computations. GPU vendors have invested significantly in developing general-purpose programming models such as CUDA, OpenCL, and OpenACC (Open Accelerators) to expose the parallel architecture of GPUs to application writers and programmers. These programming models have allowed a range of scientific applications to be accelerated on GPUs.

Figure 2.1 shows the high-level architecture of a NVIDIA GPU. It consists of streaming multiprocessors (SMs) and a memory hierarchy consisting of a large but slow global memory, a much faster on-chip shared memory, on-chip registers, cache hierarchy, and other specialized memory modules. Each SM can be seen as a vector processor, consisting of several scalar processor (SP) lanes. In the GPU memory hierarchy, global memory is the largest memory module, which is accessible to all the threads running on the GPU; however, the access latency to global memory is relatively high. Techniques, such as memory coalescing and aligned reads/writes, are often used to optimize the accesses made to global memory. Each SM of the GPU also has an on-chip shared memory, which is only accessible to the threads running on the same SM; it is much smaller in size but has much faster access times compared to global memory. In the NVIDIA Fermi architecture, the GPU memory hierarchy also has L1 and L2 caches.

Figure: 2.2 show the architecture of more recent GPUs from AMD. In contrast to the earlier VLIW architecture, as shown in Figure: 2.3, these recent AMD GPUs are based on a scalar compute architecture as part of the formally named Heterogeneous System Architecture (HSA). These newer GPUs typically have a wavefront size of 64 work-items, which are

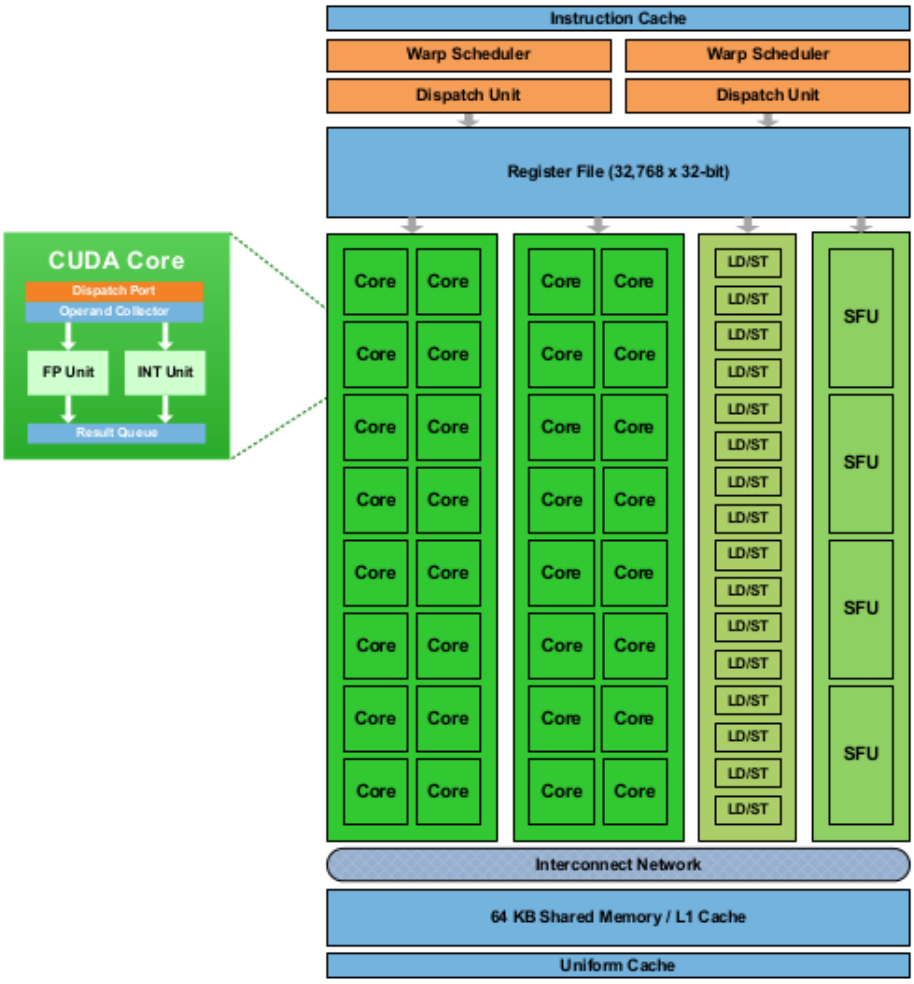


Figure 2.1: NVIDIA Fermi Architecture: Each Fermi SM includes 32 cores, 16 load/store units, four special-function units, a 32K-word register file, 64K of configurable RAM, and thread control logic. Each core has both floating-point and integer execution units. [*NVIDIAs next generation CUDA compute architecture: Fermi, white paper. Used under fair use, 2014*]

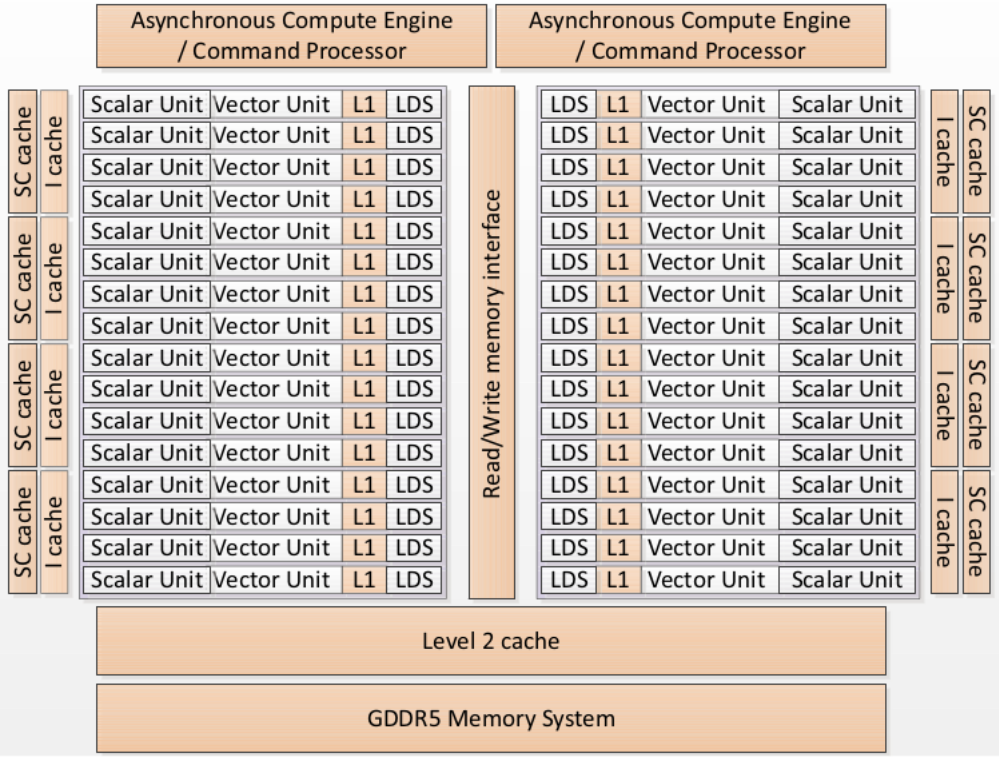


Figure 2.2: AMD 79xx Architecture: Each CU has a scalar unit, a 16-wide vector unit , L1 Cache and LDS. [*AMD Accelerated parallel processing: Opencl programming guide. Used under fair use, 2014*]

executed by a 16-wide vector unit in four cycles. These GPUs support a full-fledged cache hierarchy for instructions and data.

AMD's earlier VLIW architecture for GPUs, as realized in Evergreen and Northern Islands devices and as shown in Figure 2.3, possesses some number of "Compute Units" (CUs), in this case, four. Each CU in turn consists of 16 processing elements, and each processing element comprises four or five SIMD arrays. Every clock cycle, each of these arrays executes a single very-long instruction-word (VLIW) packet for each block of 16 work-items of a 64 work-item wavefront. These GPUs were highly dependent on compiler efficiency to extract instruction-level parallelism out of an application and to efficiently utilize the GPU unit.

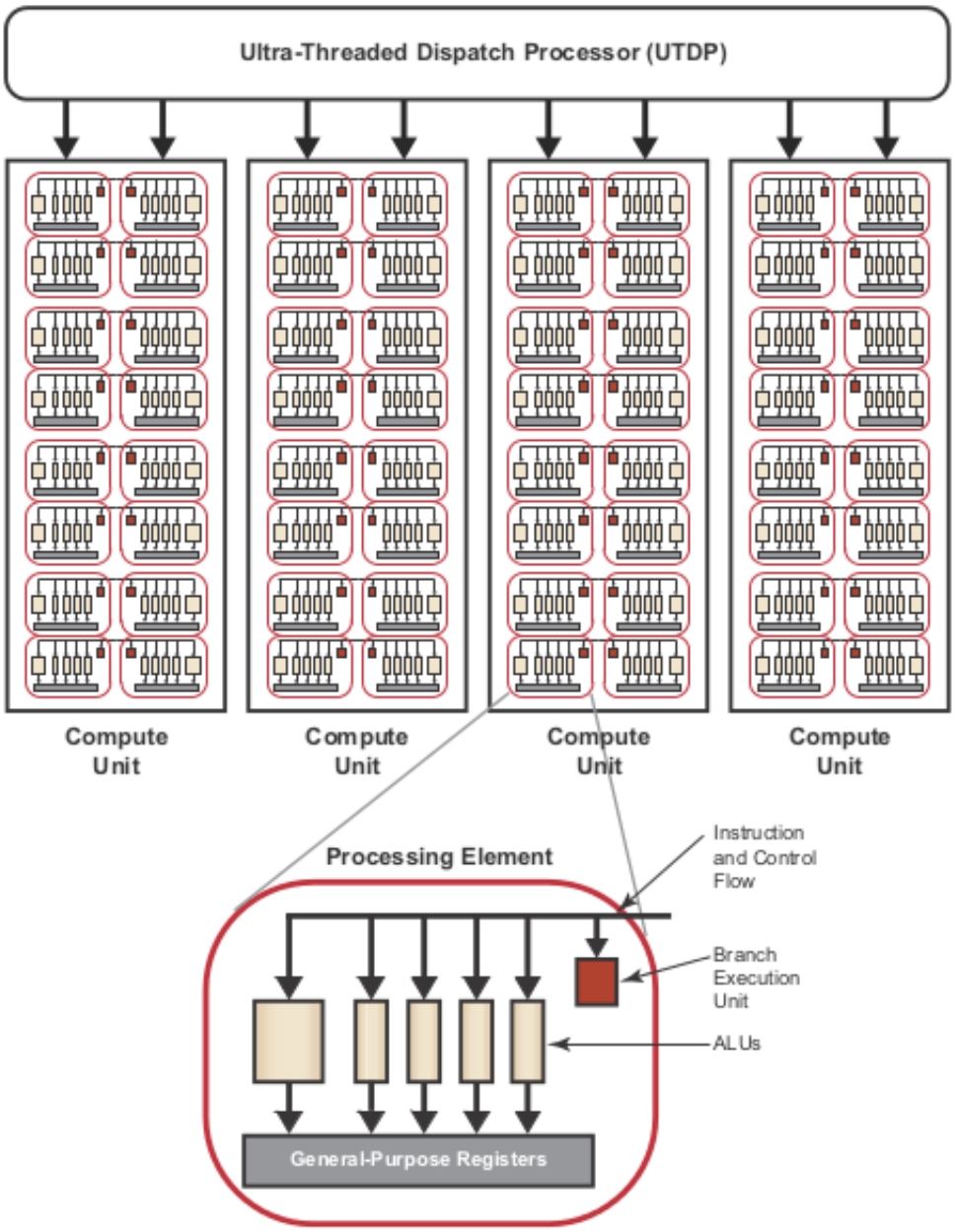


Figure 2.3: AMD Evergreen Architecture: A processing element is arranged as a 5-way VLIW processor with up to five scalar operations can be co-issued. [*AMD Accelerated parallel processing: Opencl programming guide. Used under fair use, 2014*]

2.2 Programming Models

In this subsection, we present the different programming models for GPUs at varying levels of granularity, including CUDA, OpenCL, VOCL, and MPI.

2.2.1 CUDA: Compute Unified Device Architecture

CUDA [36] provides extensions to the C/C++ programming languages to launch general-purpose computation on the GPU. The CUDA programming model requires the programmer to write the parallel regions of a code as *kernels*, which are then compiled to be launched on the GPU device. CUDA provides calls to move the data, required by the computational kernels, between the CPU and the GPU. The computational kernel primarily realizes the parallel computation with the help of blocks of threads, where the threads follow a SIMT (single instruction, multiple thread) execution paradigm.

2.2.2 OpenCL: Open Computing Language

OpenCL [22] is an open standard and parallel programming model for programming a variety of accelerator platforms, including NVIDIA and AMD GPUs, FPGAs, the Intel MIC co-processor, and conventional multicore CPUs. OpenCL follows a kernel-offloading model, where the data-parallel, compute-intensive portions of the application are offloaded from the CPU host to the device. The OpenCL kernel is a high-level abstraction with hierarchically grouped tasks or *work-groups* that contain multiple *work-items* or worker threads.

OpenCL assumes a hierarchical memory model, where the bulk of the data is assumed to be in the device's global memory and accessible by all the work-items in the kernel. Each work-group also has its own local memory, which the OpenCL implementation can map to the on-chip memory for faster accesses. The local memory is typically used to share and reuse data among threads within a work-group and cannot be accessed by any other work-group.

On the other hand, our performance projection model can be used to guide online runtime systems to choose the most suitable architecture for an already optimized user-defined kernel.

2.2.3 VOCL: Virtual OpenCL Framework

The VOCL [46] platform is a realization of the OpenCL programming model that enables the programmer to utilize both local and remote accelerators through device virtualization. The VOCL user can write a kernel for a generic OpenCL device without worrying about the physical location and other architectural details of the device. With virtual GPU frameworks such as VOCL, all the OpenCL-capable devices in a cluster can be used as if they were installed locally. The VOCL runtime system internally manages scheduling workloads to the devices in a cluster, forwarding the OpenCL calls and transferring data to and from the remote GPU. However, no clear strategy has been developed that can help such systems automatically choose the *best* GPU device for an incoming OpenCL kernel.

2.2.4 MPI: Message Passing Interface

MPI [2] continues to be one of the most widely adopted parallel programming models in scientific computing. It allows for the efficient transfer of data among different processes and address spaces across and within nodes of a cluster. MPI allows point-to-point, collective, asynchronous, and various other communication mechanisms. The traditional MPI primarily uses CPU buffers for transferring and passing data between the processes, which serves as a vehicle for one of our more important optimizations on data marshaling.

GPU-Integrated MPI

To bridge the gap between the disjointed MPI and GPU programming models, researchers have recently developed GPU-integrated MPI solutions such as our MPI-ACC [5] framework and MVAPICH-GPU [42] by Wang et al. These frameworks provide a unified MPI data transmission interface for both host and GPU memories; in other words, the programmer can use either the CPU buffer or the GPU buffer directly as the communication parameter in MPI routines. The goal of such GPU-integrated MPI platforms is to decouple the complex, low-level, GPU-specific data movement optimizations from the application logic, thus providing the following benefits: (1) portability: the application can be more portable across multiple accelerator platforms; and (2) forward compatibility: with the *same* code, the application can automatically achieve performance improvements from new GPU technologies (e.g., GPUDirect RDMA) if applicable and supported by the MPI implementation. In ad-

dition to enhanced programmability, transparent architecture specific and vendor-specific performance optimizations can be provided within the MPI layer. For example, MPI-ACC enables automatic data pipelining for inter-node communication, NUMA affinity management, and direct GPU-to-GPU data movement (GPUDirect) for all applicable intra-node CUDA communications [5,20], thus providing a heavily optimized end-to-end communication platform.

2.3 Finite Difference Method for Seismology Modeling

Our seismology application, which is based on the finite-difference method (FDM) [29], models the propagation of seismic waves using Earth's velocity structures and seismic source models as input. (For convenience, hereafter we refer to this application as *FDM-Seismology*.) The application is based on a parallel velocity-stress computation that is realized as a staggered-grid, finite-difference method (FDM), which models the propagation of waves in layered geological media. This method divides the computational domain into a three-dimensional grid and uses a one-point integration scheme for each grid cell, as shown in Figure 2.4. The application truncates the computational domain to keep the problem tractable and places absorbing boundary conditions (ABCs) around the region of interest to keep the reflections of outgoing waves minimal; hence, it helps to simulate unbounded domains. The FDM-Seismology application uses perfectly matched layers (PML) absorbers as ABCs for minimal reflection coefficient and superior efficiency.

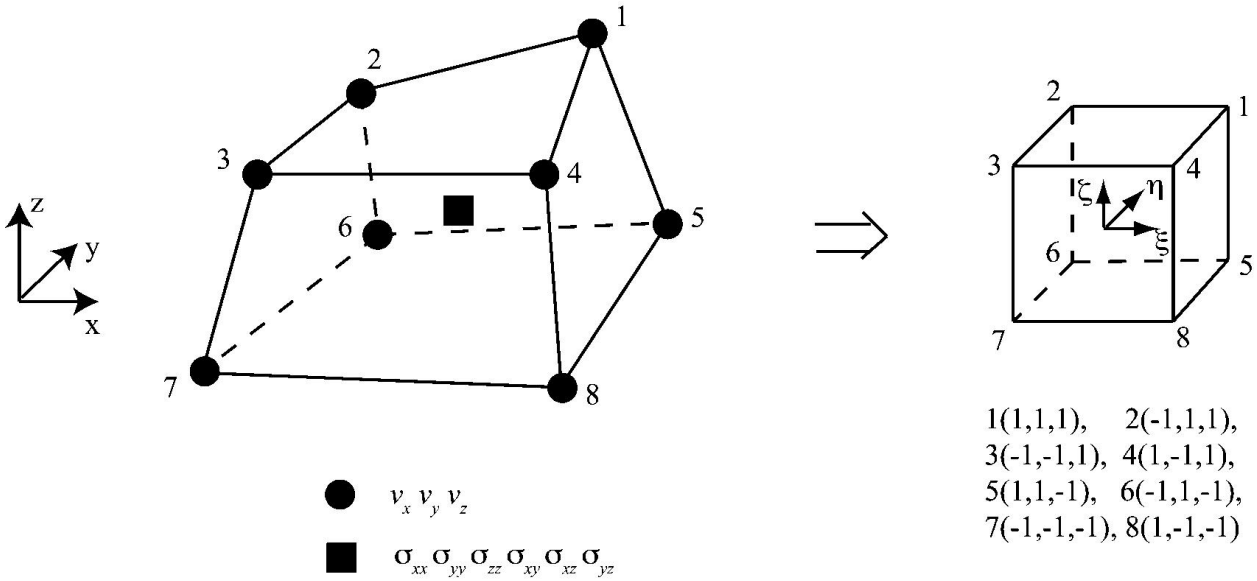


Figure 2.4: Transformation of a Hexahedreal Element into a Uniform Cube and One-Point Integration Scheme (in our one-point integration finite-difference method, all three components of velocity are defined at the nodes and the six components of stress are defined at the center of element). [*"Modeling of the perfectly matched layer absorbing boundaries and intrinsic attenuation in explicit finite-element methods." Bulletin of the Seismological Society of America. Used under fair use, 2014*]

The original implementation of the FDM-Seismology application uses domain decomposition to divide the input finite-difference model into sub-models along different axes, such that each sub-model can be computed on different CPUs (or nodes). MPI is used to distribute these sub-models across the nodes. Each node computes the velocity and stress wavefields and then exchanges the wavefields with neighboring nodes at the boundary of decomposition. These exchanges enable each processor to update its own velocity and stress wavefields so as to keep all the sub-domains consistent after each set of computations.

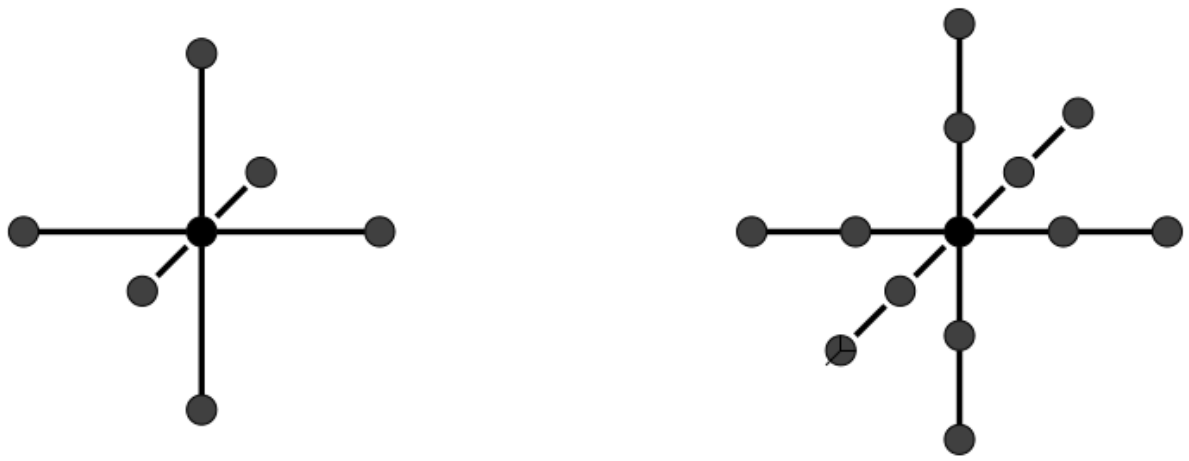


Figure 2.5: An Example of a 7-point and 13-point Stencil.

2.4 Stencil Computations

Stencil computations are iterative nearest-neighbor computations. These computations are commonly used in a wide variety of numerical and scientific computations such as electromagnetics, computational fluid dynamics (CFD), and geosciences. Stencil computations typically perform sweeps over a spatial or domain grid, where each data point in the grid is updated with weighted contributions from the neighboring elements [13]. In typical stencil computations, the grid data structures are much larger than the available cache sizes; hence, a technique called *data blocking* is used to load smaller blocks of data, serially, into the caches. The performance improvements from data blocking depend upon the relative throughput of cache and main memory as well the amount of data reuse. Figure 2.5 shows an example of a 7-point and 13-point stencil.

Chapter 3

Online Performance Projection for Heterogeneous Clusters

3.1 Introduction

Today, large-scale heterogeneous clusters predominantly consist of identical nodes in order to ease the burden of installation, configuration, and programming such systems; however, we are increasingly seeing heterogeneous clusters with different nodes on the path toward 10x10 [8], which envisions the paradigm of mapping the right task to the right processor at the right time. An early example of such a system is Darwin at LANL, which consists of both AMD and NVIDIA GPUs.

In order to improve the programmer productivity on such heterogeneous systems, virtual

GPU platforms, such as Virtual OpenCL (VOCL) [46] and Remote CUDA (rCUDA) [15], have been developed to decouple the GPU programmer's view of the accelerator resource from the physical hardware itself. For example, a user of the VOCL platform writes an OpenCL program without worrying about the physical location and other architectural details of the device. The VOCL runtime system in turn schedules and migrates application kernels among GPU devices in the backend [45].

It is critical for such runtime systems to assign the best GPU for a given kernel. Our experiments indicate that the peak performance ratios of the GPUs do not always translate to the best kernel-to-GPU mapping scheme. GPUs have different hardware features and capabilities with respect to computational power, memory bandwidth, and caching abilities. As a result, different kernels may achieve their best performance or efficiency on different GPU architectures.

However, no clear strategy exists that can help runtime systems automatically choose the best GPU device for an incoming kernel. GPU performance analysis tools have recently been developed to either predict the kernel's performance on a particular device [19] or to identify critical performance bottlenecks in the kernel [35,47]. While these techniques provide insights into the kernel and device characteristics, they are often tied to particular GPU families and require static code analysis or offline workload profiling, which makes them infeasible to be used with runtime systems.

To address this issue, we propose a technique for online performance projection, whose goal is to automatically select the best-performing GPU for a given kernel from a pool of devices.

Our online projection requires neither source-code analysis nor offline workload profiling, and its overhead in casting projections has little to no effect with increase in input sizes. We first build a static database of various device profiles; the device profiles are obtained from hardware specifications and microbenchmark characterizations and are collected just once for each target GPU.

When projecting the performance of a GPU kernel, we first obtain a workload's runtime profile using a downscaled emulation with minimal overhead, and then apply a performance model to project the full kernel's performance from the device and workload profiles. We refer to this hybrid approach of emulation and modeling as *mini-emulation*. The emulation code is invoked by intercepting GPU kernel launch calls, an action that is transparent from the user's perspective. Our technique is particularly suitable for projecting the relative performance among GPU devices for kernels with large datasets.

Our specific contributions are as follows:

- An online, kernel-characterization technique that retrieves key performance characteristics with relatively low overhead by emulating a downscaled kernel execution.
- An online performance model that projects the performance of full kernel execution from statistics collected from the downscaled emulation.
- An end-to-end implementation of our technique that covers multiple architectural families from both AMD and NVIDIA GPUs.

We evaluate our online performance projection technique by ranking four GPUs that belong to various generations and architecture families from both AMD and NVIDIA. Our workload consists of a variety of memory- and compute-bound benchmarks from the AMD APP SDK [6]. We show that the device selection recommended by our technique is the best-performing device in most cases.

3.2 Related Work

Several techniques for understanding and projecting GPU performance have been proposed. These techniques are often used for performance analysis and tuning. We classify the prior work into two categories: performance modeling and performance analysis and tuning.

Performance Modeling GPU performance models have been proposed to help understand the runtime behavior of GPU workloads [19, 47]. Some studies also use microbenchmarks to reveal architectural details [19, 41, 44, 47]. Although these tools provide in-depth performance insights, they often require static source analysis or offline profiling of the code by either running or emulating the program.

Several cross-platform, performance-projection techniques can be used to compare multiple systems, many of them employ machine learning [28, 39]. However, the relationship between tunable parameters and application performance is gleaned from profiled or simulated data.

Performance Analysis and Tuning Researchers have proposed several techniques to analyze GPU performance from various aspects, including branching, degree of coalescing, race conditions, bank conflict, and partition camping [4, 9, 38]. They provide helpful information for the user to identify potential bottlenecks.

Several tools have also been developed to explore different transformations of a GPU code [24, 31]. Moreover, Ryoo et al. proposed additional metrics (efficiency and utilization) [37] to help prune the transformation space. Furthermore, several autotuning techniques have been devised for specific application domains [10, 32].

The various techniques either require the source code for static code analysis or rely on the user to manually model the source-code behavior; both approaches are infeasible for application to a runtime system. Moreover, they often require offline profiling, which may take even longer than the execution time of the original workload. To our knowledge, no GPU performance models have been developed that are suitable for online device selection.

3.3 Design

The goal of online performance projection is two-fold: (1) to accurately rank the GPU devices according to their computational capabilities and (2) to do so reasonably quickly in support of dynamic runtime scheduling of GPUs. The actual execution of the kernel on the target GPUs serves as the baseline to evaluate both the accuracy and the performance of any performance projection technique. However, for a runtime system in cluster environments,

it is infeasible to always run the kernel on all the potential devices before choosing the best device, because of the additional data transfer costs. Below we discuss the accuracy vs. performance tradeoffs of potential online performance projection techniques for GPUs.

Cycle-accurate emulation, with emulators such as GPGPU-Sim [3] and Multi2Sim [40], can be used predict the minimum number of cycles required to execute the kernel on the target device. The accuracy of the projection and the device support directly depends on the maturity of the emulator. Moreover, the overhead of cycle-accurate emulation is too high to be used in a runtime system.

Static kernel analysis and projection can be done at (1) the *OpenCL code* level, (2) an *intermediate GPU language* level (e.g., PTX or AMD-IL), or (3) the *device instruction* level (e.g., cubin). Performance projection from static analysis will be inaccurate because it does not take into account the dynamic nature of the kernel, including memory access patterns and input data dependence. The performance projection will, however, not experience much overhead and is feasible to be used at runtime.

Dynamic kernel analysis and projection involve a tradeoff between the above approaches. The dynamic kernel characteristics, such as instruction distribution, instruction count, and memory access patterns, can be recorded by using functional emulators, such as GPU-Ocelot [14] or the functional emulator mode of GPGPU-Sim and Multi2Sim. The dynamic kernel profile, in conjunction with the target device profile, can be used to develop a performance projection model. While the accuracy of this approach is better than that of static code analysis, the emulation overhead of this approach will be greater. On the other hand,

the overhead will be much smaller than with cycle-accurate emulation.

3.3.1 Approach

We realize a variant of the dynamic kernel analysis for performance projection while significantly limiting the emulation overhead with minimal loss in accuracy of projection. Our online projection technique requires that all the target GPU devices are known and accessible and that the workload's input data is available. However, we can intercept OpenCL's kernel setup and launch calls to obtain the required OpenCL kernel configuration parameters.

As Figure 3.1 shows, our online projection consists of three steps. First, we obtain the hardware characteristics through device characterization. The capability of the device may vary according to the *occupancy*[1] or the *device utilization* level. We use microbenchmarks to profile the devices and their efficiency in response to different occupancy levels.

Second, we collect the dynamic characteristics of an incoming kernel at runtime. We leverage existing GPU emulators and develop a miniature emulator that emulates only one work-group of the kernel. With such an approach, we can obtain dynamic workload characteristics including instruction mix, instruction count, local and global memory accesses, and the coalescing degree. The code for our mini-emulator is invoked by transparently intercepting GPU kernel launch calls.

Third, with the per work-group characteristics and the per device hardware profile, we project

[1]Occupancy refers to the ratio of active wavefronts to the maximum active wavefronts supported by the GPU.

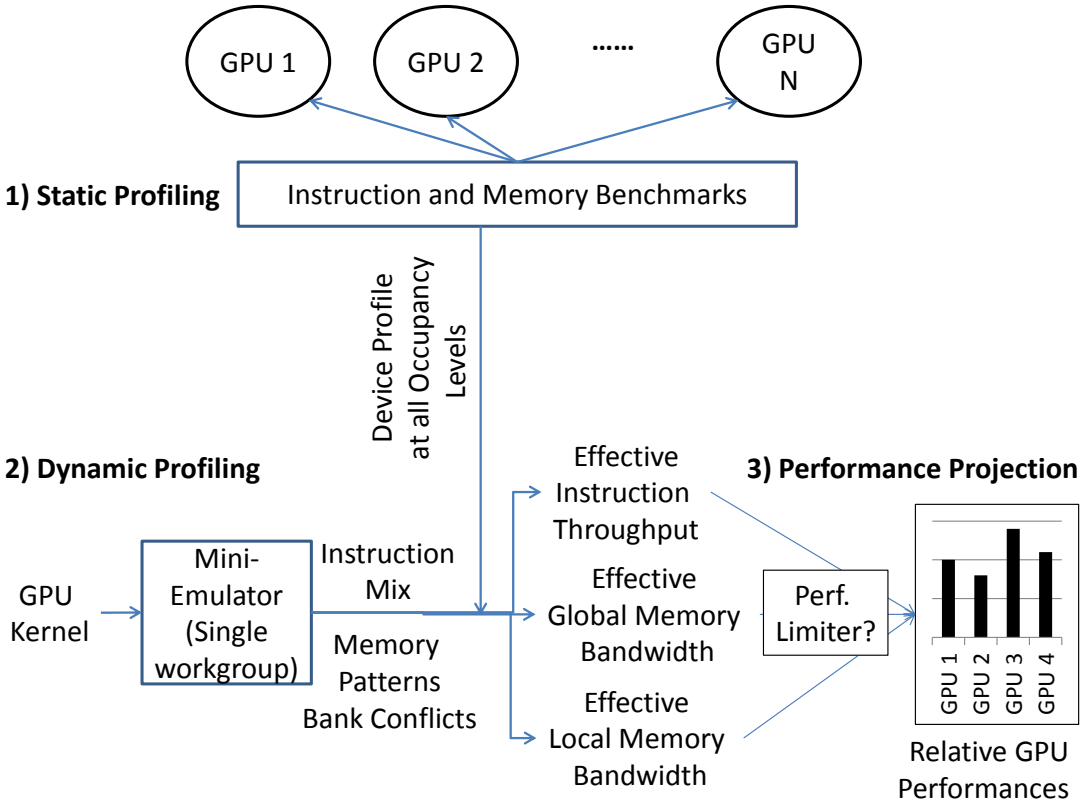


Figure 3.1: The Performance Projection Methodology.

the runtime of the full kernel execution. Our projection also takes into account various potential performance limiting factors and compares the tradeoffs among devices. GPU runtime systems, such as VOCL, can then select the ideal performing device for the purposes of migration of an already running workload for consolidation or scheduling subsequent invocations in case of repeated execution.

3.3.2 Device Characterization

Our device characterization focuses on three major components of GPU performance: instruction throughput, local memory throughput, and global memory throughput. The in-

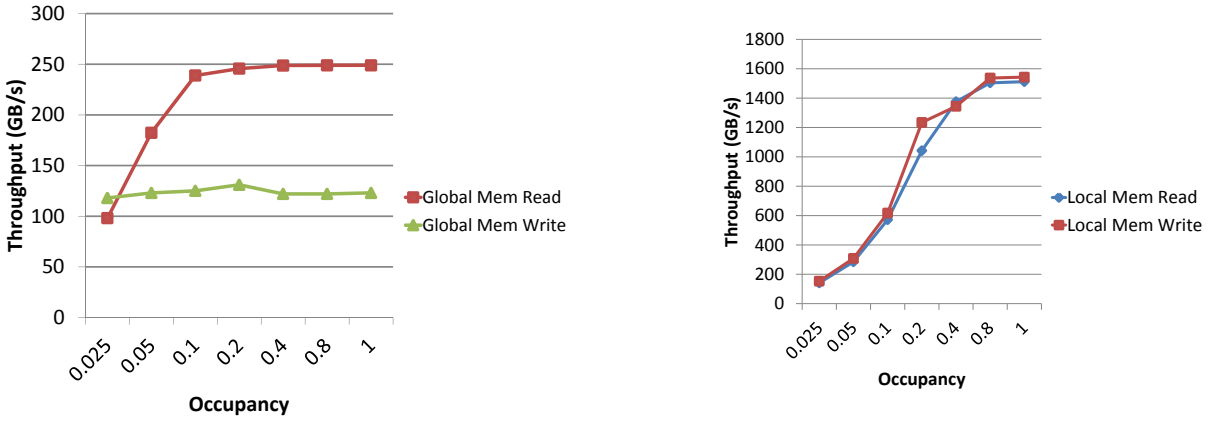


Figure 3.2: Global and Local Memory Throughput on the AMD HD 7970.

struction throughput of a device (or peak FLOP rate) can be obtained from hardware specifications, and the memory throughput under various occupancy levels and access patterns are measured through microbenchmark profiling. We use microbenchmarks derived from the SHOC benchmark suite [12] to measure the hardware's dynamic memory performance under different runtime scenarios, such as occupancy, type of the memory accessed, and word sizes. The hardware characteristics are collected only once per device.

For the global memory accesses, the microbenchmarks measure the peak throughput of coalesced accesses for read and write operations at various occupancy levels. For example, Figure 3.2 shows the coalesced memory throughput behavior for the AMD HD 7970 GPU. The throughput for uncoalesced memory accesses are derived by analyzing the coalesced throughputs along with the workload characteristics that are obtained from the emulator, as described in Section 3.3.4.

Similar to global memory, the local memory benchmarks measure the throughput of local memory at varying occupancy levels of GPU (Figure 3.2). Our local memory throughput microbenchmarks do not account for the effect of bank conflicts, but our model deduces the number of bank conflicts from the emulator's memory traces and adjusts the projected performance as described in Section 3.3.4.

3.3.3 Online Workload Characterization

This subsection describes our fully automated approach for dynamically obtaining a workload's characteristics—in particular, statistics for both dynamic instructions and dynamic memory accesses—in order to cast performance projections.

Statistical measures about dynamic instructions include the instruction count, branch divergence intensity, instruction mixes, and composition of the very long instructions. The dynamic memory access statistics include the local and global memory transaction count, bank conflict count, and the distribution of coalesced and uncoalesced memory accesses. The above runtime characteristics can impact the actual kernel performance in different ways on different GPUs. For example, the HD 5870 is more sensitive to branching than the HD 7970 [6]. Similarly, the NVIDIA C1060 has fewer shared memory banks and is more sensitive to bank conflicts than the C2050. Emulators are useful tools to obtain detailed workload characteristics without extensive source code analysis. However, the time to emulate the entire kernel is usually orders of magnitude larger than the time to execute the kernel itself.

Therefore, off-the-shelf emulators are not suitable for online projection.

Mini-Emulation

To alleviate the emulation overhead problem, we propose a technique named "mini-emulation", which employs a modified emulator that functionally emulates just a single work-group when invoked. Our assumption is that work-groups often exhibit similar behavior and share similar runtime statistics, which is typical of data-parallel workloads. Subsequently, the aforementioned runtime statistics of the full kernel can be computed from the number of work-groups and the statistics of a single work-group, thereby significantly reducing the emulation overhead.

To emulate both NVIDIA and AMD devices, we adopt and modify two third party emulators: GPGPU-Sim [3] for NVIDIA devices and Multi2sim [40] for AMD devices. We note that our technique can employ other emulators as long as they can generate the necessary runtime characteristics and support the OpenCL frontend. Our modified mini-emulators accept a kernel binary and a set of parameters as input, emulates only the first work-group, ignores the remaining work-groups and outputs the appropriate statistics. We do not change the task assignment logic in the emulator, i.e. the single emulated work-group still performs the same amount of work as if it were part of the full kernel having many work-groups.

Deriving Full Kernel Characteristics

The mini-emulator outputs characteristics of only a single work-group. To cast performance projections, however, we need to obtain the characteristics of the full kernel. The scaling factor between the characteristics of a single work-group and that of a full kernel depends on the device occupancy, which in turn depends on the per-thread register usage and local memory usage, which can be obtained by inspecting the kernel binary.

Using device occupancy as the scaling factor, we linearly extrapolate statistics about dynamic instructions and memory accesses of a single work-group to that of the full kernel. The derived characteristics of the full kernel can then be used to project the kernel's performance.

3.3.4 Online Relative Performance Projection

The execution time of a kernel on a GPU is primarily spent executing compute instructions and reading and writing to the global and the local memory. Hence, we follow an approach similar to [47] in modeling three relevant GPU components for a given kernel: compute instructions, global memory accesses and local memory accesses. Moreover, GPUs are designed to be throughput-oriented devices that aggressively try to hide memory access latencies, instruction stalls and bank or channel conflicts by scheduling new work. So, we assume that the execution on each of the GPU's components will be completely overlapped by the execution on its other components, and the kernel will be bound only by the longest running component. We then determine the bounds of a given kernel for all the desired GPUs and

project the relative performances. While our three component based model is sufficiently accurate for relative performance projection, it is easily extensible to other components such as synchronization and atomics for higher levels of desired accuracy. We will now describe the approach to independently project the execution times of the three GPU components.

Compute Instructions ($t_{compute}$) When the given OpenCL workload is run through the emulator, our model obtains the total number of compute instructions and calculates the distribution of instruction types from the instruction traces. The throughput for each type of instruction can be found in the GPU vendor manuals. We model the total time taken by the compute instructions as $\sum_i (\frac{instructions_i}{throughput_i})$ where i is the instruction type.

Global Memory Accesses (t_{global}) The global memory performance of a GPU kernel can be affected by the memory access patterns within a wavefront, because the coalescing factor can influence the total number of read and write transactions made to the global memory. For example, in an NVIDIA Fermi GPU, a wavefront (containing 32 work-items or threads) can generate up to thirty two 128B transactions for completely uncoalesced accesses, but as low as a single transaction if all the accesses are coalesced and aligned. Hence, there can be up to a 32-fold difference in the bandwidth depending on the coalescing factor on the Fermi. From the memory access traces generated from the emulators, we can deduce the coalescing factor and the number of memory transactions generated per wavefront. Since the memory transaction size and coalescing policies vary with each GPU, we calculate the number of transactions using device-specific formulas. Since the throughput of global memory also

varies with the device occupancy, we inspect our per-device characteristics database and use the throughput value at the given kernel's occupancy. We model the time taken by global memory accesses as $\frac{trasactions \times transaction\_size}{throughput_{occupancy}}$. We calculate the above memory access times separately for read and write transactions and sum them to obtain the total global memory access time.

Local Memory Accesses (t_{local}) The local memory module is typically divided into banks and accesses made to same banks are serialized. On the other hand, accesses made to different memory banks are serviced in parallel to minimize the number of transactions. We inspect the GPU emulator's local memory traces and calculate the degree of bank conflicts. Also, we use the local memory throughput at the given kernel's occupancy for our calculations. We calculate the total time taken by the local memory accesses similar to that of global memory, where we model the read and write transactions separately and sum them to get the total local memory access time.

The boundedness of the given kernel is determined by the GPU component that our model estimates to be the most time consuming, i.e. $max(t_{compute}, t_{global}, t_{local})$.

3.4 Evaluation

In this section, we describe our experimental setup and present the evaluation of our online performance projection model.

3.4.1 System Setup

Our experimental setup consists of four GPUs: two from AMD and two from NVIDIA. We used the AMD driver v9.1.1 (fglrx) for the AMD GPUs and the CUDA driver v285.05.23 for the NVIDIA GPUs. The host machines of each GPU were running 64-bit Linux. We used Multi2sim v4.0.1 for simulating the OpenCL kernels on AMD devices and GPGPU-Sim v3 for simulating the NVIDIA GPUs.

Table 3.1 presents the architectural details of the GPUs. Besides the differences in the number of computation units and memory modules, these devices also represent a variety of GPU architectures. While the AMD HD 5870 is based on the previous VLIW-based 'Evergreen' architecture, the AMD HD 7970 belongs to the new Graphics Core Next (GCN) 'Southern Islands' architecture, where it moves away from the VLIW-based processing elements (PEs) to scalar SIMD units. The architecture of HD 7970 closely resembles the NVIDIA C2050 (Fermi) architecture in terms of the scalar SIMD units and the presence of a hardware cache hierarchy. The key differences between the NVIDIA C1060 'Tesla' and the NVIDIA C2050 'Fermi' architectures are the hardware cache support, improved double-precision support, and dual-wavefront scheduling capabilities on the newer Fermi GPU.

Table 3.2 summarizes our chosen set of eight benchmarks from the AMD APP SDK v2.7, which are all written in OpenCL v1.1. We chose benchmarks that exhibited varying computation and memory requirements.

We apply our model-based characterization, as described in Section 3.3.4, to identify the

Table 3.1: Summary of GPU Devices.

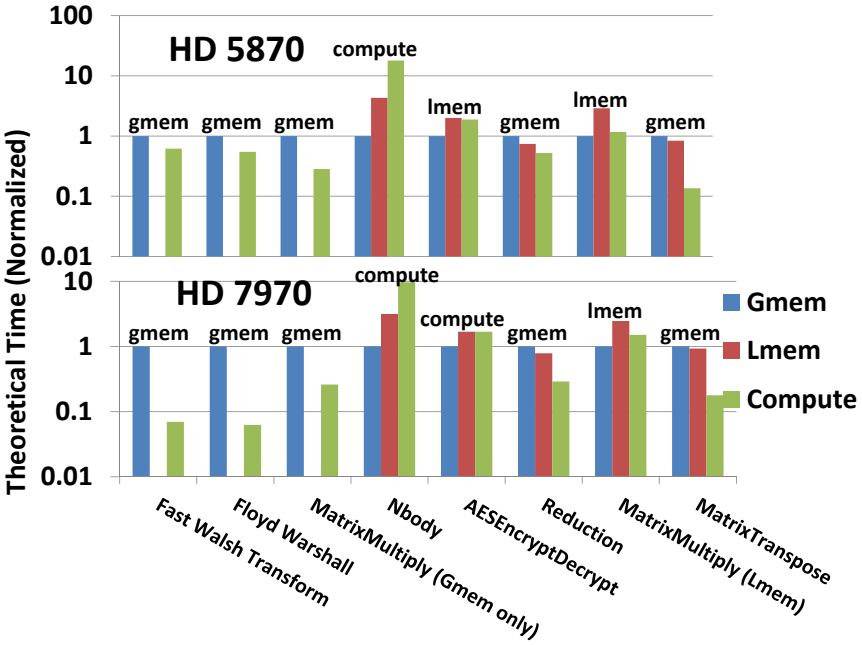
GPU	Arch. Name	Compute Units	Peak Perf. (GFlops)	Peak Mem. BW (GB/s)	Mem. Transaction Sizes (B)	Shared Mem. Banks
HD5870	Evergreen	20	2720	264	64	32
HD7970	Southern Islands	32	3790	154	64	32
C1060	Tesla	30	933	102	32,64,128	16
C2050	Fermi	14	1030	144	128	32

Table 3.2: Summary of Applications.

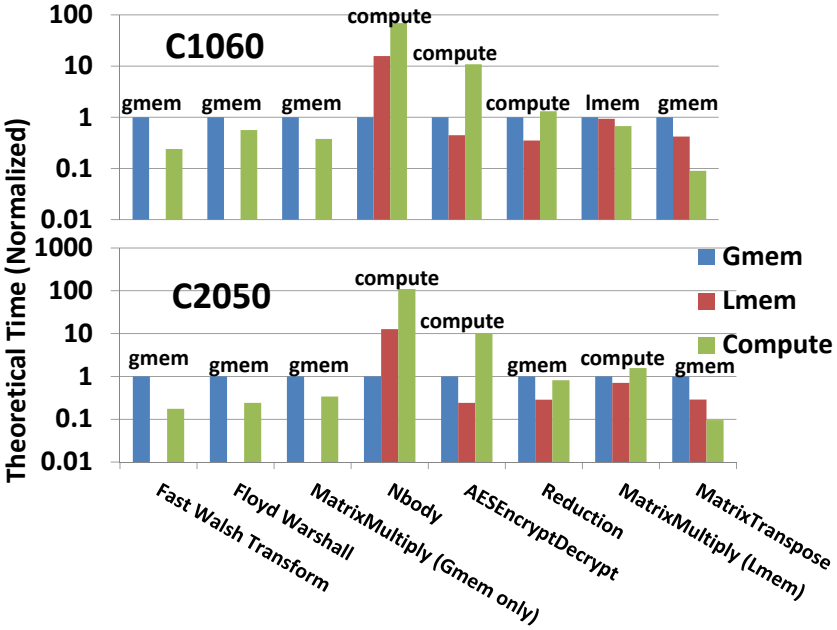
FloydWarshall	FastWalshTrans.	MatMul (gmem)	MatMul (lmem)
Nodes = 192	Size = 1048576	[1024,1024]	[1024,1024]
Reduction	NBody	AESEncryptDecr.	Matrix Transpose
Size = 1048576	Particles = 32768	W=1536, H=512	[1024,1024]

performance bottlenecks in computation, global memory throughput, and local memory throughput. Figure 3.3 presents the projected normalized execution times for the AMD and NVIDIA devices and characterizes the applications by the performance limiting components.

Figure 3.3 shows that our set of benchmarks exhibits a good mix of application characteristics, where the benchmarks are bounded by different GPU components. Our model establishes that the benchmarks FloydWarshall, FastWalsh, MatrixTranspose, MatrixMultiply (global memory version), and NBody retain their boundedness across all the GPUs, while the boundedness of other applications changes across some of the devices. Figure 3.3 also suggests that the performance limiting components of AESEncryptDecrypt and MatrixMultiply (local memory version) are not definitive because the projected times are very close and within the error threshold. For example, AESEncryptDecrypt can be classified as either



(a) Application Analysis of AMD GPUs.



(b) Application Analysis of NVIDIA GPUs.

Figure 3.3: Determining the Performance Limiting Factor (Gmem, Lmem, or Compute) for Different Applications. At the top of each bar, we note the limiting component for an application on the device.

local memory bound or compute bound for the AMD devices, and MatrixMultiply (local memory version) can be local memory or global memory bound for the NVIDIA C1060. However, the ambiguity in boundedness does not affect the relative ranking of the GPUs because our model just picks one of the competing components as the performance limiter.

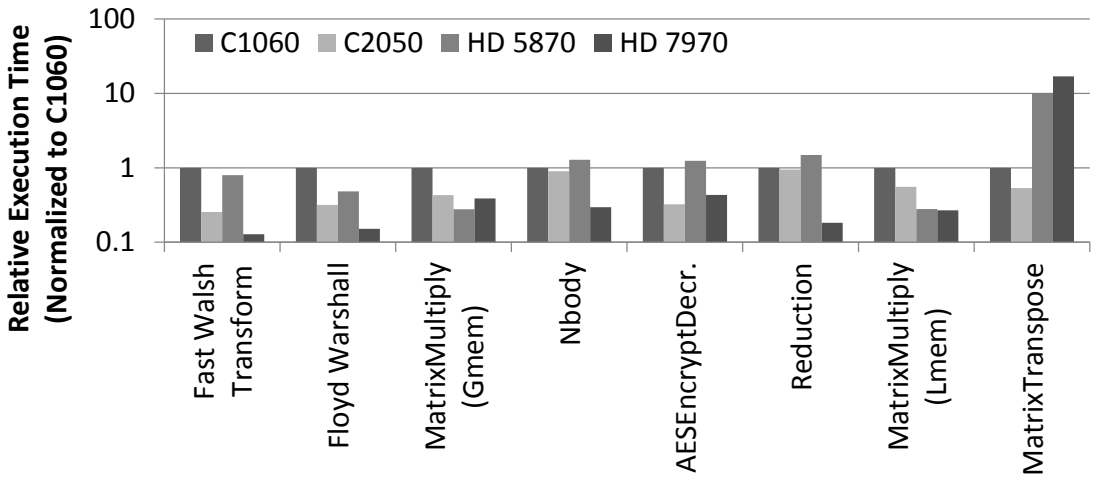
3.4.2 Results

Our online performance projection technique needs to estimate the relative execution time among devices within a small amount of time. Therefore, we evaluate our technique from two aspects: modeling accuracy and modeling overhead.

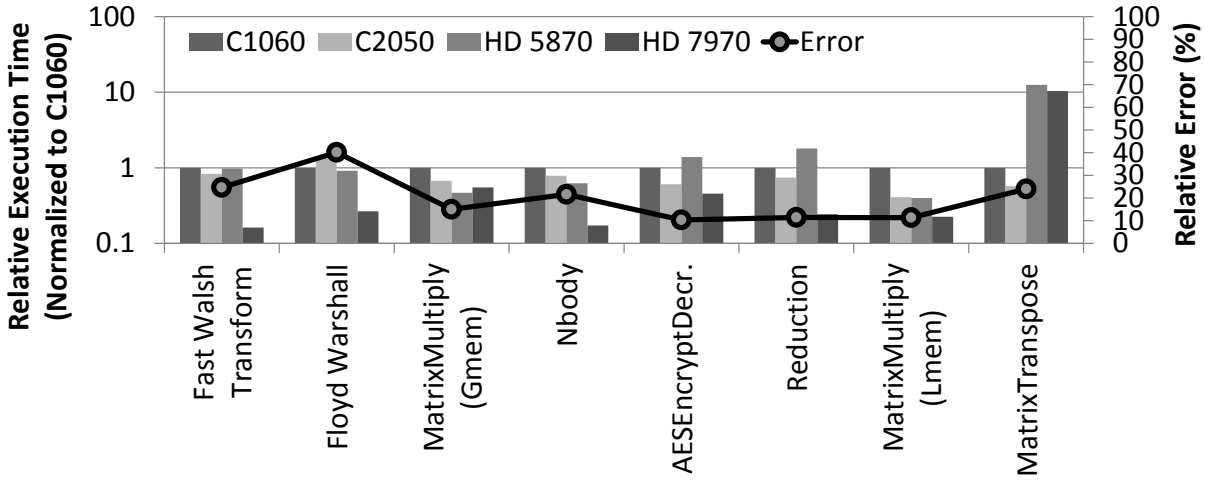
Accuracy of Performance Projection

In this section, we evaluate the ability of our technique to project the relative performance among target GPUs. Figure 3.4a shows the actual execution time of all benchmarks on our four test GPUs, and Figures 3.4b and 3.4c show the projected execution times by the single work-group mini-emulation and the full kernel emulation, respectively. All the numbers are normalized to the performance of NVIDIA C1060.

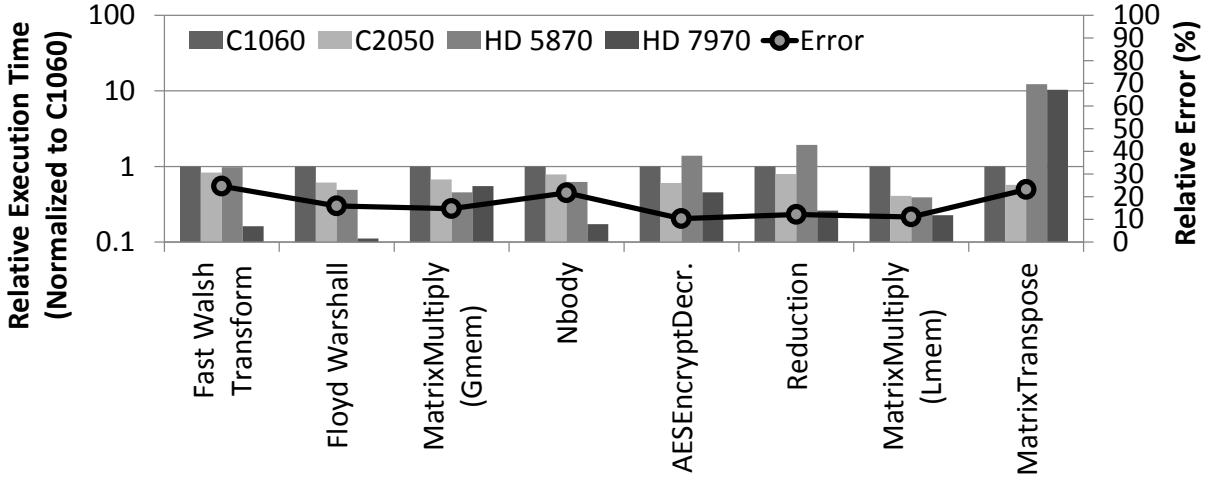
Selecting the Best Device The main purpose of our performance projection model is to help runtime systems choose the GPU that executes a given kernel in the smallest amount of time. We define the *device selection penalty* to be the $\frac{|T(B)-T(A)|}{T(A)} \times 100$, where $T(A)$ is the runtime of the kernel over its best performing GPU and $T(B)$ is the runtime of the kernel



(a) Actual Execution Times.



(b) Projected Execution Times by Single Work-group Mini-Emulation.



(c) Projected Execution Times by Full Kernel Emulation.

Figure 3.4: Accuracy of the Performance Projection Model.

over the recommended GPU.

Figure 3.4 shows that our model picks the best device for all cases except one: the AES-Encrypt application. In this case, our model picks the AMD HD 7970, whereas the best device is C2050, with a device selection penalty of 33.72%.

Relative Performance Among Devices In some cases, the runtime system may want to perform a global optimization to schedule multiple kernels over a limited number of GPUs. In those circumstances, the runtime system may need information about the kernel's relative performance among the GPUs in addition to the best GPU for each kernel. This helps the runtime system evaluate the tradeoffs of various task-device mappings and make judicious decisions in scheduling multiple kernels.

We measure the error of the relative performance as follows. Let us consider the kernel's actual performance on the four GPUs as one 4D vector, T_{actual}. Similarly, the kernel's projected performance on the four GPUs can then be represented as another 4D vector, $T_{projected}$. T'_{actual} and $T'_{projected}$ are the normalized, unit-length vectors for T_{actual} and $T_{projected}$, respectively. They reflect the relative performance among the GPUs. We then formulate the error metric of our relative performance projection to be $\frac{||T'_{actual} - T'_{projected}||}{\sqrt{2}} \times 100\%$, which ranges from 0% to 100% and correlates with the Euclidean distance between T'_{actual} and $T'_{projected}$. Note that $\sqrt{2}$ is the maximum possible Euclidean distance between unit vectors with non-negative coordinates. For the single work-group mini-emulation mode, the average relative error of our model across all kernels in our benchmark suite is 19.8%, with the relative

errors for the individual applications ranging from 10% to at most 40%. On the other hand, if the full-kernel emulation mode is used, then the average relative error becomes 16.7%.

Limitation of Our Performance Projection Model We note that the single work-group mini-emulation mode does not change the individual application-specific relative errors from the full kernel emulation for most of the applications, with the exception of Floyd Warshall. A key requirement for the mini-emulation mode is that all the work-groups of the kernel must be independent of each other and that all the work-groups will execute in approximately the same amount of time with the same number of memory transactions and instructions. The Floyd Warshall application comprises a series of converging kernels, where the input of one kernel is dependent on the output of the previous kernel; that is, there is data dependence between iterations. Since only a single work-group is being emulated in the mini-emulation mode, all the data elements are not guaranteed to be updated by the program iterations, thereby causing the memory and compute transactions to change over iterations. Since Floyd Warshall is a global memory-bound application, we compared the projected global memory transactions of the mini- emulation mode and the full kernel emulation modes for similar global memory bound kernels, as shown in Figure 3.5. We see that for the Floyd Warshall application, the projected global memory transactions using the mini-emulation mode is 2.5 times less than the projected transactions from the full kernel emulation mode for the C1060 GPU. For the C2050 GPU, this difference is less: 10%. The data-dependent and iterative nature of the Floyd Warshall application introduces errors into

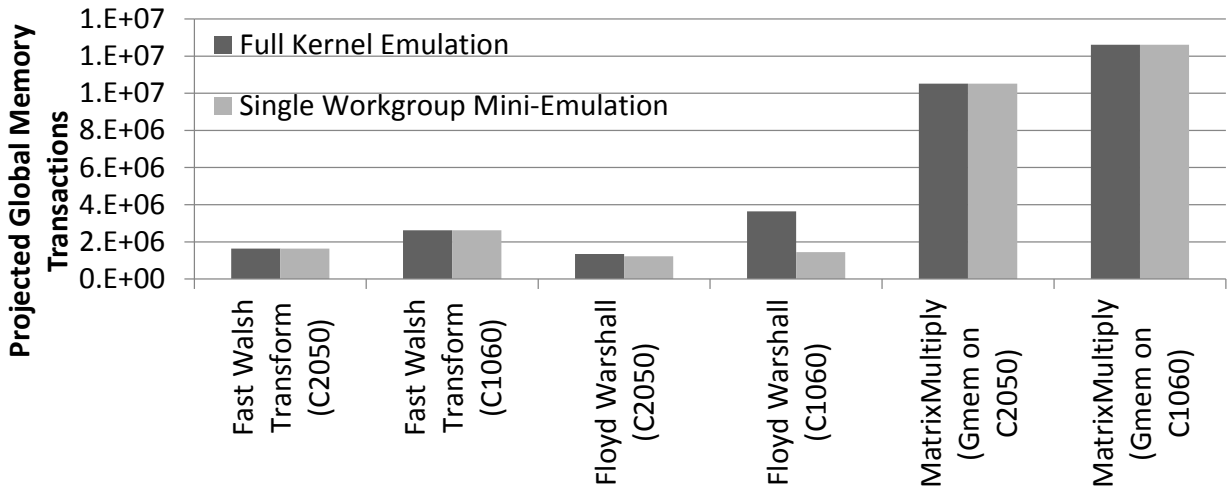


Figure 3.5: Global Memory Transactions for Select Applications.

our mini-emulation–based projection model, which may cause our model to pick the wrong GPU in some cases.

Overhead of Performance Projection

The overhead of our online projection includes time spent in online workload characterization as well as casting the projected performance for each device. Because hardware characterization is done only once for each hardware, it does not incur any overhead at the time of projection. Among the two sources of overhead, casting performance projection need only calculate a few scalar equations; it has a constant and negligible overhead. The major source of overhead comes from the online workload characterization using mini-emulation, which functionally emulates one work-group to collect kernel statistics.

The state-of-the-art technique to obtain detailed runtime statistics of a kernel is full kernel emulation. As Table 3.3 shows, our mini-emulation approach reduces the emulation overhead

Table 3.3: Performance Model Overhead Reduction – Ratio of Full-Kernel Emulation Time to Single Work-group Mini-Emulation Time.

Application	Fast Walsh Trans-form	Floyd Warshall	MatMul (Gmem)	Nbody	AES Encrypt	Reduction	MatMul (Lmem)	Matrix Trans-pose
C1060	2882.1	172.7	267.9	3.9	710.4	1710.7	240.7	554.2
C2050	2772.7	182.6	337.4	4.1	784.0	1709.0	196.6	556.1
HD 5870	2761.0	248.5	219.1	4.7	623.4	2629.1	201.5	469.2
HD 7970	2606.9	229.3	217.7	4.0	581.4	2700.0	209.6	457.1

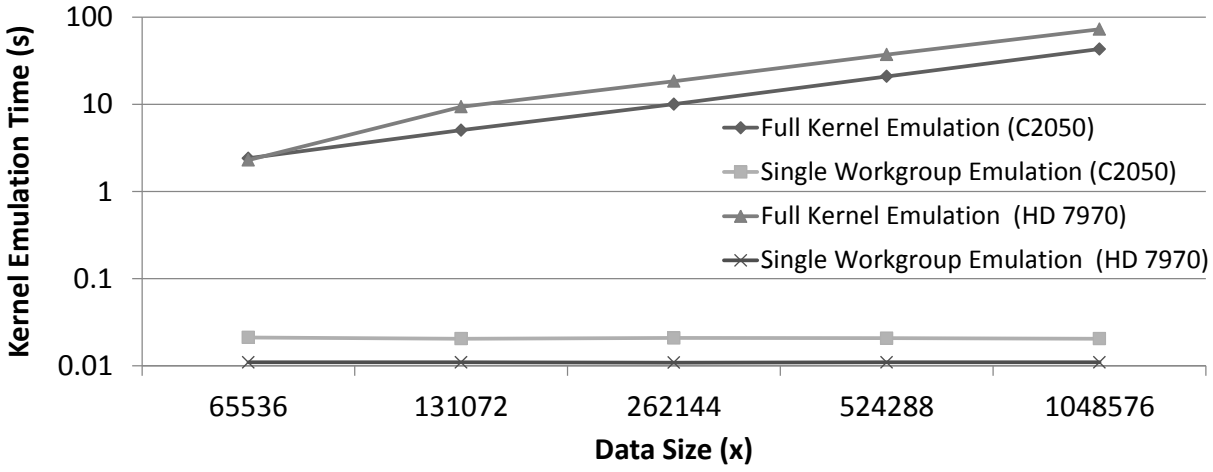
by orders of magnitude. Meanwhile, it obtains the same level of details about runtime characteristics. In fact, the mini-emulation overhead is often comparable to kernel execution time with small or moderate-sized inputs and will be further dwarfed if the kernel operates over a large dataset, as is often the case for systems with virtual GPU platforms. Such a low overhead makes it worthwhile to employ our technique to schedule kernels with large datasets; it also allows the runtime system to evaluate the task-device mapping in parallel with the workload execution, so that it can migrate a long running workload in time. Below we further study the relationship between input data size and the overhead of mini-emulation.

Impact of Input Data Sizes Figure 3.6 shows the performance impact of the input data size on the full-kernel emulation and single work-group mini-emulation overheads. Figure 3.6a shows that the full-kernel emulation overhead for the reduction kernel increases with data size. The reduction kernel launches as many work-items as the data elements, thereby having a one-to-one mapping between the work-item and the data element. As the data size grows, the number of work-items also increases, so that each work-item and work-group has

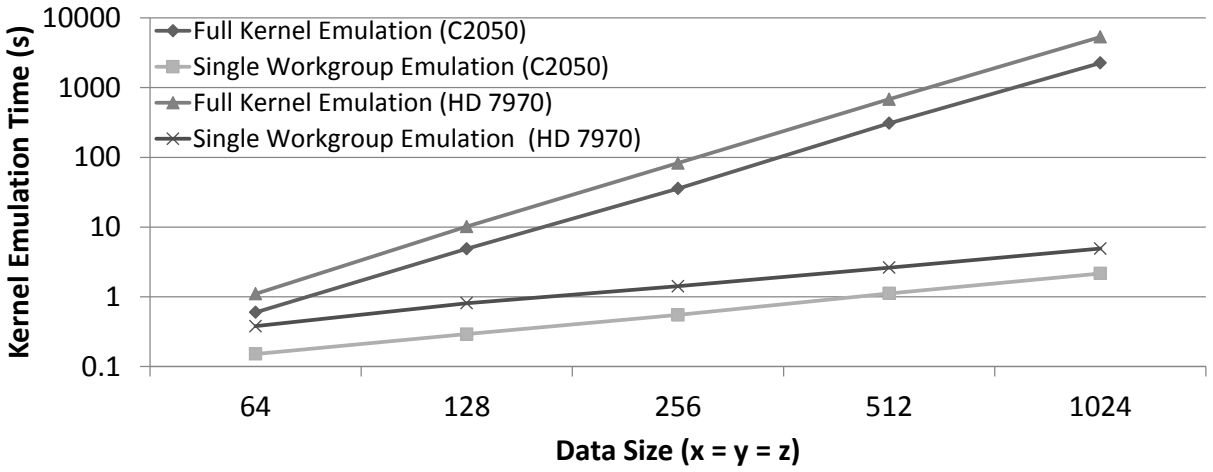
a constant amount of computation. Since each work-item of the kernel is simulated sequentially on the CPU, the overhead of the full-kernel emulation also increases for larger data sizes. On the other hand, our mini-emulation scheme simulates just a single work-group to collect the kernel profile irrespective of the number of data elements. That is why we see a constant overhead for the mini-emulation scheme for reduction kernel.

Figure 3.6b shows that both the full kernel emulation overhead and the mini-emulation overhead for the matrix multiplication kernel increases with data size. However, we observe that the rate of increase of overhead (slope of the line) is linear for the mini-emulation mode, while it is cubic for the full kernel emulation mode. The matrix multiply kernel launches as many work-items as the output matrix size, but unlike the reduction kernel, each work-item and work-group do not have a constant amount of computation. The load on each work-item increases linearly with the matrix length (we choose only square matrices for the sake of simplicity). This is why we see that the mini-emulation overhead increases linearly with the matrix length for the matrix multiplication kernel.

However, the actual GPU execution time itself increases in a cubic manner with the matrix length; thus, our mini-emulation mode is asymptotically better and will take less time than running the kernel on the device for larger data sizes. Figure 3.7 shows that as the matrix length increases, the GPU execution time approaches the kernel mini-emulation time for the NVIDIA C2050 GPU. We were unable to store even larger matrices on the GPU's 3 GB global memory; but we can infer that for even larger matrix sizes, our mini-emulation technique will outperform the actual GPU execution. On the other hand, if the mini-emulation time



(a) Kernel: Reduction.



(b) Kernel: Matrix Multiplication (Using Local Memory).

Figure 3.6: Kernel Emulation Overhead – Full Kernel Emulation vs. Single Work-group Mini-Emulation.

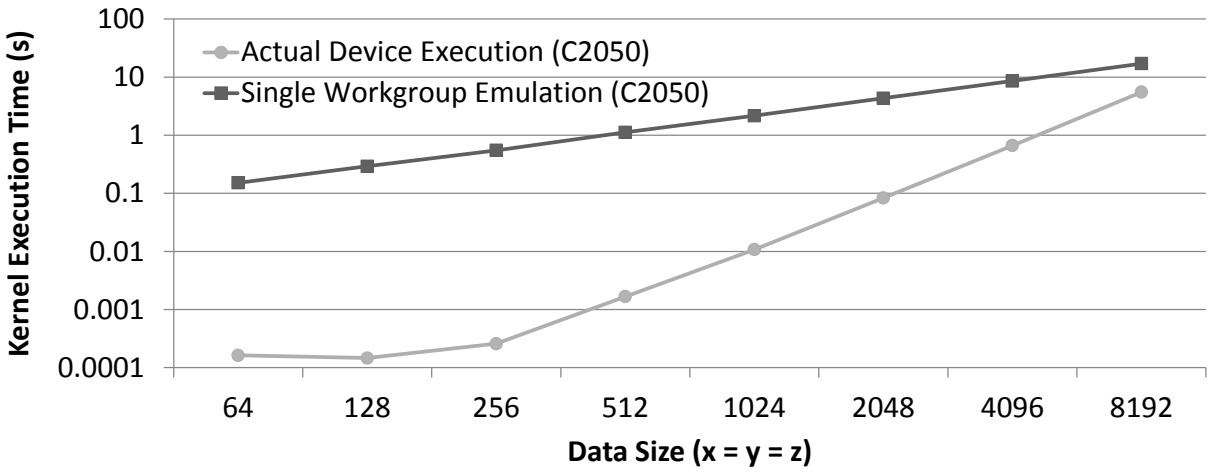


Figure 3.7: Kernel Emulation Overhead – Single Work-group Mini-Emulation vs. Actual Device Execution.

remains constant, as with the reduction kernel, then it is obvious that the mini-emulation approach will cause the least overhead for larger data sizes, thereby making our model amenable to dynamic decision-making runtime systems.

Chapter 4

Performance Optimization and Characterization of FDM-based Seismology Simulation

4.1 Introduction

Modeling the behavior of seismic wave propagation inside the subsurface of Earth is often complicated by factors such as heterogeneity of the composition of geological layers, irregular geometries, and anisotropy. Using analytical methods to investigate the behavior of these waves is not particularly useful because such methods are formulated upon tractable assumptions of regular geometries, homogeneity, and isotropy. Numerical meth-

ods, on the other hand, are based upon discrete approximations of the wave equations, and they do approximately account for the heterogeneity of realistic geometries and other important physical properties, such as stress and friction. Among these numerical methods, the finite-difference method (FDM) is one of the most popular methods, primarily because of its simplistic implementation and relatively better computational efficiency. The FDM primarily uses a grid-based scheme to approximate the solutions of partial differential equations. The method divides the problem domain into a grid of points, and the derivatives of partial differential equations are approximated by linear combinations of function values at the grid-points.

However, FDM in seismological simulations is computationally intensive, and there have been many past efforts to accelerate these computations on large clusters as well as heterogeneous systems, consisting of CPUs and GPUs. GPUs are massively parallel computing processors and have gained widespread popularity as general-purpose accelerators in the scientific community. However, accelerating any application on GPUs often requires various architecture-aware optimizations to achieve significant speedup; which otherwise can be severely limited by factors such as uncoalesced memory access patterns, divergent branches, and expensive data-transfers over the PCIe interface.

Our FDM-based seismology application, hereafter referred as *FDM-Seismology*, is a MPI+CUDA realization of an application that models the propagation of seismological waves using the finite-difference method by taking the Earth's velocity structures and seismic source models as input [29]. The application implements a parallel velocity-stress computation, based on

a staggered-grid, finite-difference method [11, 17, 30], for propagation of waves in a layered medium. In this method, the domain is divided into a three-dimensional grid, and a one-point-integration scheme [29] is used for each grid cell. Since the computational domain is truncated in order to keep the computation tractable, absorbing boundary conditions (ABCs) are placed around the region of interest so as to keep the reflections minimal when boundaries are impinged by the outgoing waves [29]. This strategy helps simulate unbounded domains. In our application, absorbers based on perfectly matched layers (PML) [7] are used as ABCs for their superior efficiency and minimal reflection coefficient. The use of a one-point integration scheme leads to an easy and efficient implementation of the PML-absorbing boundaries and allows the use of irregular elements in the PML region [29].

In this chapter, we present strategies to map FDM-Seismology onto a CPU-GPU heterogeneous cluster. We then describe how performance models can be used to explain and help drive application-specific optimizations. We discuss the scalability of the application over a large number of nodes and the efficacy of using dual GPUs per node in accelerating the application. We also discuss how a GPU-integrated library, MPI-ACC, allows our application design to evolve to a much simplified, improved, and performance-efficient design by providing a heavily optimized and direct communication channel between remote GPU memory spaces.

4.2 Related Work

Many seismology-modeling codes have been ported to large-scale computing clusters. With the advent of GPUs as hardware accelerators in such large-scale clusters, many recent efforts have sought to accelerate their seismological codes on the GPU. Our FDM-Seismology code seeks to do the same. However, the underlying numerical methods in our model possess different characteristics from those in existing literature [25–27, 33]. These previous efforts have used finite element or finite difference with time domain analysis, along with different boundary-absorbing conditions to model the seismic waves. In contrast, our realization uses a staggered-grid, finite-difference method (FDM) with PML-absorbing boundaries. In this section, we will discuss other efforts and their underlying numerical algorithms used to model seismological wave propagation.

Komatitsch et al. [26] have ported a high-order, finite- element method onto a heterogeneous cluster and have reported the overall speedup to be 20.1 over the serial version; their computations are based on an unstructured mesh and use mesh coloring to handle the summation operations. Michéa et al. [33] demonstrates a simulation that is based on finite difference in time-domain wave propagation and convolution-based, perfectly-matched boundaries to absorb outgoing waves. The simulation uses NVIDIA Tesla GPUs to accelerate the code within a compute node and MPI for communication between nodes in a cluster. To accelerate these computations on a large number of cluster nodes, Komatitsch et al. [27] model the propagation of elastic waves on 192 GPUs and achieve a per-node speedup of 20.6-fold,

relative to two CPU cores. Komatitsch et al. [25] also used a different numerical method called spectral-element method to model seismic wave propagation through both fluid and solid layers.

4.3 Design

In this section, we discuss the design of our GPU-accelerated FDM-Seismology application, including intra-node optimizations, inter-node optimizations, and dual-GPU optimizations.

4.3.1 Intra-node Optimizations

The FDM-Seismology application primarily consists of the velocity and stress computations that generate large wavefields, which are stored as multi-dimensional, floating-point arrays. Each set of velocity and stress computations is then followed by wavefield exchanges with neighboring nodes. The velocity and stress computations are the most computationally intensive components of FDM-Seismology and account for over 97% of the execution time on a CPU. Hence, the first step towards porting FDM-Seismology to the GPU is to accelerate the velocity and stress computations on the GPU, given their relatively data-parallel computational pattern.

Like many scientific applications, FDM-Seismology is implemented in Fortran. Rather than relying on third-party compilers such as the PGI CUDA Fortran compiler, we chose to

maintain compatibility with the more supported CUDA compiler for C, provided by NVIDIA, and ported the velocity and stress computation functions from Fortran to C. However, the differences in data layout between the two programming models needed to be carefully considered as it could significantly impact performance when the functions or *kernels* run on the GPU.

Specifically, Fortran uses *column-major* ordering of multi-dimensional arrays, which flattens the arrays column-wise such that the first dimension is contiguous in the memory. However, C uses *row-major* ordering, which stores the elements of a multi-dimensional array to make the elements of the last dimension contiguous. We keep the front-end of our application, which deals with files to read input and write output, in Fortran; so, the multi-dimensional arrays are read into memory in column-major order. Because the memory accesses made by a warp on NVIDIA Fermi GPUs are coalesced into as few as cache lines as possible, threads that access global memory in contiguous fashion are much more efficient in utilizing the memory bandwidth. In our case, however, the arrays are stored in column-major order in the GPU memory, so this implementation accesses memory in an *uncoalesced* fashion, thus resulting in significantly worse performance, a well-known problem in GPU code optimization. However, rather than introduce the overhead of transposing the arrays from column-major to row-major in order to make the memory accesses coalesced, we refactored the loops inside the computational kernels, using a technique called *loop permutation*, to change the memory access pattern from uncoalesced to coalesced.

We now discuss the ways to efficiently pre-stage these computations into the shared memory

of a GPU, which is an on-chip memory module and has much higher memory throughput.

Blocking Data Layout The computational pattern in FDM-Seismology is effectively a stencil computation. In our case, it is a 7-point 3D stencil computation, similar to that shown in Figure 2.5, where each data point accesses two additional neighbors in the three dimensions, while computing a 3D sub-domain on the GPU. We launch one CUDA thread for each data point in the sub-domain. Hence, each thread has to access the data from six additional points for its local computation. Because the same data points are reused by multiple neighboring threads in the stencil computation, we pre-stage these data domains into the shared memory of the GPU, which can deliver 8-10 times higher bandwidth than the global memory of the GPU.

However, due to the limited shared memory available to each SM on the GPU, the sub-domain is further decomposed into "data blocks," which are serially loaded into shared memory, as part of the technique called *blocking*. This blocking can be performed in two ways: 2D blocking or 3D blocking. The 2D-blocking approach loads data as 2D planes, as shown in Figure 4.1, which allows for maximum reuse of data in the 2D plane. The 3D blocking allows reuse of data in all the three dimensions, but it suffers from the drawback of loading more boundary cells in a constrained and limited shared memory space.

In this chapter, we use performance modeling to explain the efficacy of different blocking strategies and predicting the performance impact without an actual execution of the program on the GPU. We believe that such performance models are important as they enable scientists

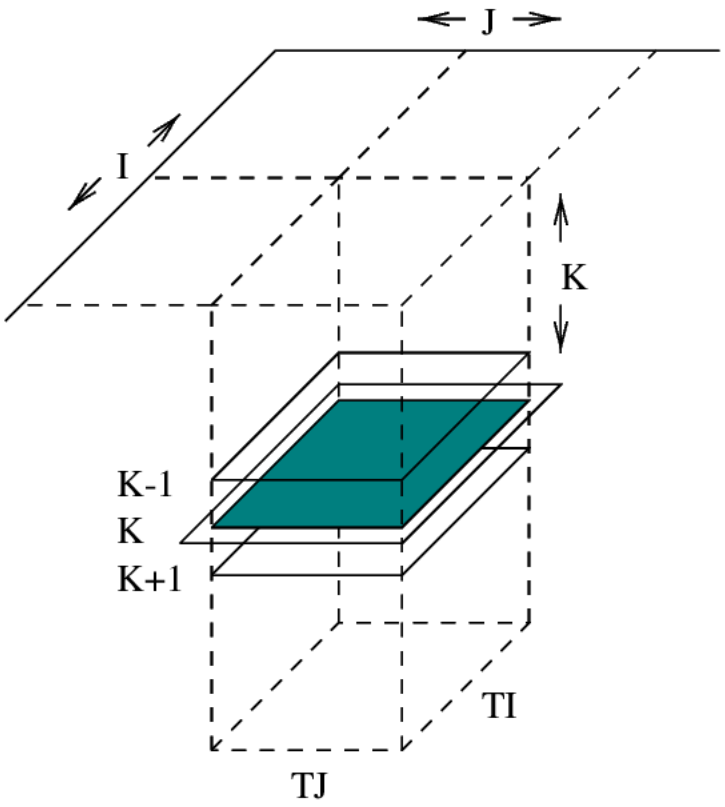


Figure 4.1: An Example of 3D Grid Data being Blocked as 2D Planes for a 7-point Stencil (it includes two planes of 'halo cells', along k direction, and a boundary of 'halo cells' around the block plane). [*"Tiling optimizations for 3d scientific computations". In Supercomputing, ACM/IEEE 2000 Conference. Used under fair use, 2014*]

to experiment with the efficacy of various optimization strategies without the need of an actual deployment.

4.3.2 Inter-node Optimizations

The simulation operates on the input finite-difference (FD) model and generates a three-dimensional grid as a first step. Our MPI-based parallel version of the application divides the input FD model into submodels along different axes such that each submodel can be

computed on different nodes. This technique, also known as domain decomposition, allows the computation in the application to scale to a large number of nodes. Each processor computes the velocity and stress wavefields in its own subdomain and then exchanges the wavefields with the nodes operating on neighbor subdomains, after each set of velocity or stress computation (Figure 4.2a). These exchanges help each processor update its own wavefields after receiving wavefields generated at the neighbors.

These computations are run for a large number of iterations for better accuracy and convergence of results. In every iteration, each node computes the velocity components followed by the stress components of the seismic wave propagation. The wavefield exchanges with neighbors take place after each set of velocity and stress computations. This MPI communication takes place in multiple stages wherein each communication is followed by an update of local wavefields and a small postcommunication computation on local wavefields. At the end of each iteration, the updated local wavefields are written to a file.

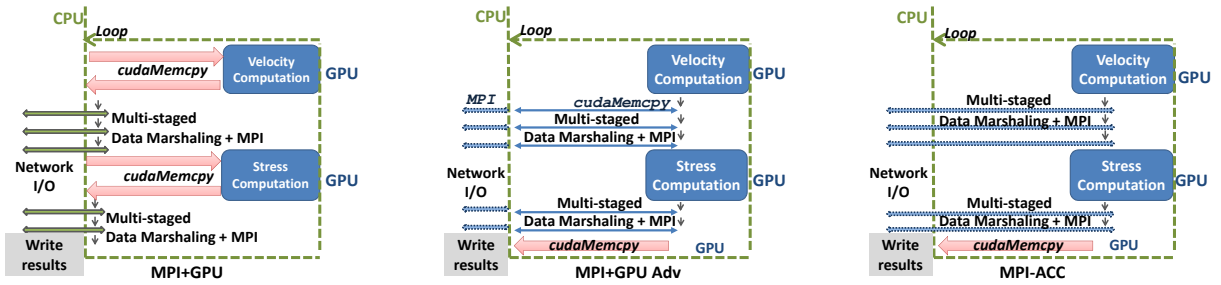
The velocity and stress wavefields are stored as large multidimensional arrays on each node. In order to optimize the MPI computation between neighbors of the FD domain grid, only a few elements of the wavefields, those needed by the neighboring node for its own local update, are communicated to the neighbor, rather than whole arrays. Hence, each MPI communication is surrounded by data marshaling steps, where the required elements are packed into a smaller array at the source, communicated, then unpacked at the receiver to update its local data.

GPU Acceleration of FDM-Seismology

MPI+GPU with data marshaling on CPU (MPI+GPU): Our GPU-accelerated version of the application performs the velocity and stress computations as kernels on the GPU. In order to transfer the wavefields to other nodes, it first copies the bulk data from the GPU buffers to CPU memory over the PCIe bus and then transfers the individual wavefields over MPI to the neighboring nodes, as shown in Figure 4.2a. All the data marshaling operations and small postcommunication computations are performed on the CPU itself. The newly updated local wavefields that are received over MPI are then bulk transferred back to the GPU before the start of the next stress or velocity computation on the GPU.

MPI+GPU with data marshaling on GPU (MPI+GPU Adv): A much faster GDDR5 memory module is available on the GPU. If the memory accesses are coalesced, the data marshaling module performs much better than the CPU, which has the slower DDR3 memory. Also, the packing and unpacking operations of the data marshaling stages can benefit from the highly multithreaded SIMT execution nature of GPU. Hence, it is a natural optimization to move the data marshaling operations to the GPU, as shown in Figure 4.2b. Moreover, the CPU-GPU bulk data transfers that used to happen before and after each velocity-stress computation kernel are avoided. The need to explicitly bulk transfer data from the GPU to the CPU arises only at the end of the iteration, when the results are transferred to the CPU to be written to a file.

However, such an optimization has the following disadvantage in the absence of GPU-



(a) Basic MPI+GPU FDM-Seismology Application with Data Marshaling on CPU.

(b) MPI+GPU FDM-Seismology Application with Data Marshaling on GPU.

(c) MPI-ACC–driven FDM-Seismology Application with Data Marshaling on GPU.

Figure 4.2: Communication-Computation Pattern in the FDM-Seismology Application.

integrated MPI. All data marshaling steps are separated by MPI communication, and each data marshaling step depends on the preceding marshaling step *and* the received MPI data from the neighbors. In other words, after each data marshaling step, data has to be explicitly moved from the GPU to the CPU only for MPI communication. Similarly, the received MPI data has to be explicitly moved back to the GPU before the next marshaling step. In this scenario, the application uses the CPU only as a communication relay. If the GPU communication technology changes (e.g., GPU-Direct RDMA), we will have to largely rewrite the FDM-Seismology communication code to achieve the expected performance.

MPI-ACC–enabled Optimizations

Since the MPI-ACC library enables MPI communication directly from the GPU buffer, the new version of the application retains the velocity and stress computation results on the GPU itself, performs the packing and unpacking operations using multiple threads on the GPU, and communicates the packed arrays directly from the GPU. Similar to the MPI+GPU Adv

case, the bulk transfer of data happens only once at the end of each iteration, when results are written to a file.

The MPI-ACC–driven design of FDM-Seismology with data marshaling on the GPU greatly benefits from the reduction in the number of expensive synchronous bulk data transfer steps between the CPU and GPU. Also, since the data marshaling step happens multiple times during a single iteration, the application needs to launch a series of marshaling kernels on the GPU. While consecutive kernel launches entail some kernel launch and synchronization overhead per kernel invocation, the benefits of faster data marshaling on the GPU and optimized MPI communication make the kernel synchronization overhead insignificant.

GPU-driven data marshaling provides the following benefits to MPI+GPU Adv and MPI-ACC–based designs of FDM-Seismology: (1) it removes the need for the expensive bulk `cudaMemcpy` data transfers that were used to copy the results from the GPU to the CPU after each set of velocity and stress computations; and (2) the application benefits from the multiple threads performing the data packing and unpacking operations in parallel, and on the faster GDDR5 device memory.

Other than the benefits resulting from GPU-driven data marshaling, a GPU-integrated MPI library benefits the FDM-Seismology application in the following ways: (1) it significantly enhances the productivity of the programmer, who is no longer constrained by the fixed CPU-only MPI communication and can easily choose the appropriate device as the communication target end-point; (2) the pipelined data transfers within MPI-ACC further improve the communication performance over the network; and (3) regardless of the GPU communication

technology that may become available in the future, our MPI-ACC–driven FDM-Seismology code will not change and will automatically enjoy the performance upgrades that are made available by the subsequent GPU-integrated MPI implementations (e.g., support for GPU-Direct RDMA).

Dual GPU Optimizations

Since FDM-Seismology is an MPI based application, it can be scaled to multiple GPUs on each node by launching a MPI process for each GPU. Our evaluation cluster has two GPUs per node, hence we also study the intra-node multi-GPU scaling of the FDM-Seismology.

4.4 Evaluation

4.4.1 System Setup

We conducted our experiments on a 455-teraflop (single precision) hybrid CPU-GPU supercomputer. Each node of the cluster contains two CPU sockets, each containing a hex-core 2.4 GHz Intel Xeon E5645 CPU. Each node also has two NVIDIA Tesla M2050 GPUs. The total host memory on each node is 24 GB and each GPU has 3 GB of GDDR5 memory. Each of the two GPUs is connected to separate CPU socket via PCIe express. The interconnect between the nodes is QDR InfiniBand. Our compiler suite consists of GCC v4.7 as the C compiler, gfortran v4.7 as the Fortan-90 compiler, and CUDA compiler v4.0 with driver

version 270.41.34.

4.4.2 Intra-node Optimizations

Kernel Optimizations

In this subsection, we describe the effects of the optimizations used to accelerate the FDM computations on the GPU. As discussed earlier, the serial version of the FDM code, written in Fortran, makes uncoalesced access to the global memory due to the memory layout of arrays in Fortran. Hence, the straightforward GPU implementation of the code results in each thread making uncoalesced global memory accesses. Hence, the first optimizing step is to convert the pattern of memory accesses made by threads in a warp to coalesced accesses. Figure 4.3 shows that we gained 16.3-fold speedup by converting the uncoalesced memory accesses to coalesced accesses.

The FDM memory accesses also follow a stencil-based computation pattern where neighboring cells of the grid communicate with each other during the computation. Hence, we leverage the fast on-chip shared memory resource on the GPU to cache the memory accesses made to the global memory. Figure 4.3 shows that we gain an additional 1.5-fold speedup by using the shared memory for each block of threads. The low speedup gained suggests low data-reuse in the FDM-based computations.

The data transfers to the GPU are also overlapped with the computation from asynchronous launches of kernel computation. However, the overlap only occurs at the beginning of the

velocity and stress computations; hence the additional speedup gains are limited to 1.8-fold.

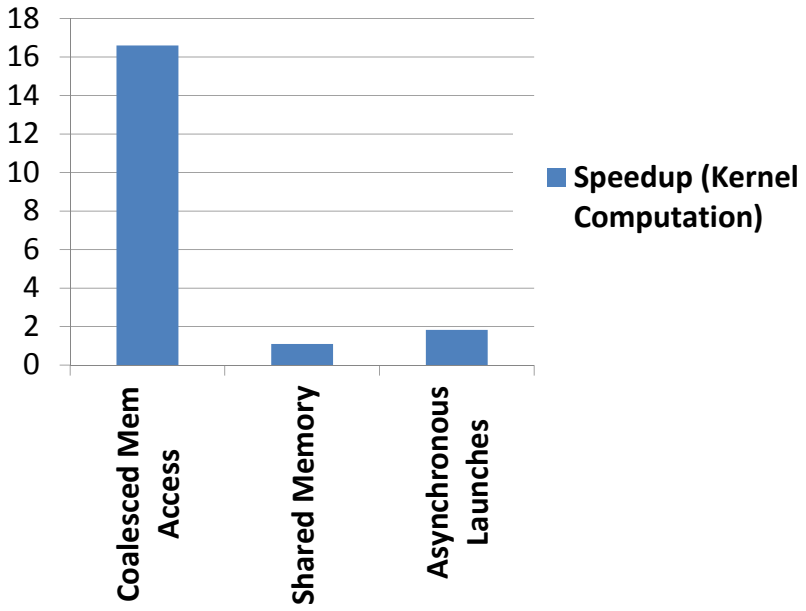


Figure 4.3: Intra-node GPU Optimizations for FDM-Seismology.

Data Transfer Optimizations

As noted earlier, the data transfers over PCI Express (PCIe), which occur before and after each set of velocity and stress computations, present a major bottleneck in extracting performance from the GPU. Figure 4.4 shows that a large part of the execution time is spent in moving the data between the GPU and the CPU. This serial component in the overall execution reduces the total achievable speedup to just 5.2-fold.

However, when the data-marshaling operations (i.e., packing and unpacking the wavefield parameters) are ported to the GPU, they benefit from its superior parallel-processing capabilities as well as the avoidance of costly bulk transfers that occur at the end of each set of

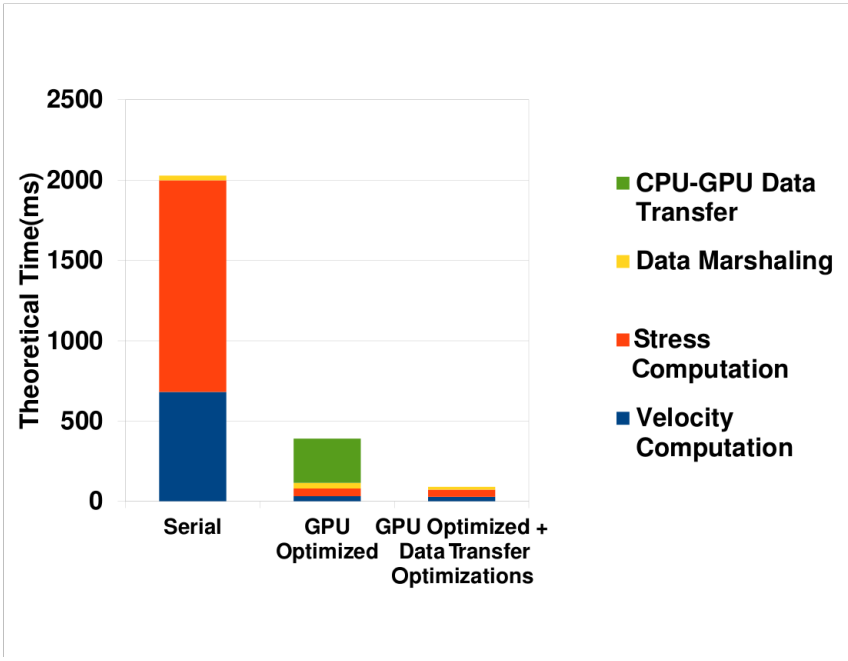


Figure 4.4: Performance Improvements with Data Transfer Optimizations.

velocity and stress computations. In all, by moving the data-marshaling computations to the GPU, we achieve an overall speedup of 23-fold over the serial version of the application running on the CPU on a single core.

Dual-GPU Scaling

The FDM-based modeling of seismic waves is based upon a 3D-domain decomposition technique, which allows a 3D-domain to be decomposed into multiple sub-domains and where each sub-domain can be computed within a separate MPI process.

In addition to using MPI to scale the computation across multiple nodes, we further scale the computation within a node by using dual GPUs in each node. Specifically, each cluster node has two GPUs, hence, we launch two MPI processes per node, with each MPI process

utilizing one GPU for acceleration of its own sub-domain. We obtain a peak speedup of 33-fold on a single node with two GPUs, as shown in Figure 4.5. The speedup obtained from dual GPUs decreases rapidly when the computation is weakly scaled; primarily because per-node computation decreases much faster to compensate for the data-transfer costs.

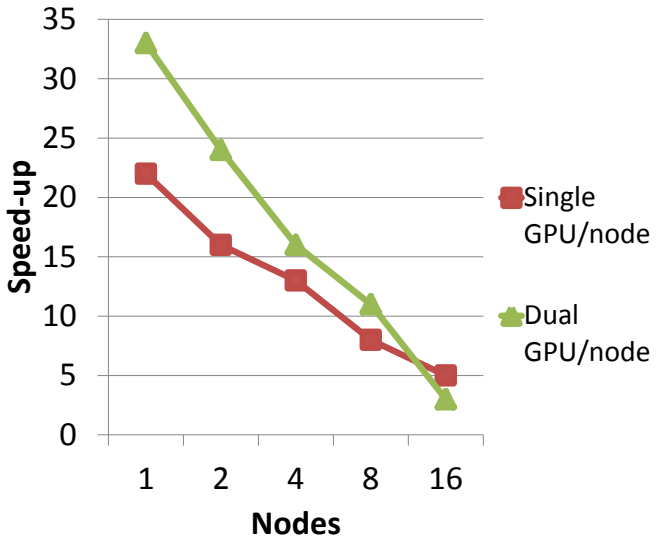


Figure 4.5: Weak Scaling with Single and Dual GPUs Per Node.

4.4.3 Performance Characterization

In this section, we describe how we use the performance modeling technique, as described in the Chapter 3 (Section 3.3.4), to reason about the results of the optimizations. At the same time, the correlation between the theoretical and empirical results affirms that the performance model can be used to drive the optimizations offline, before being deployed to the actual cluster resource.

We simulate a single iteration of FDM-Seismology computation in the GPGPU-Sim emulator

to obtain the dynamic characteristics of the computation; in particular, statistics related to compute instructions and accesses made to global and shared memory. These dynamic characteristics lead us to the evaluation of $t_{compute}$, t_{shared} and t_{global}, the maximum of which determines the performance bottleneck, as described in the Chapter 3 (Section 3.3.4).

Preliminary Performance Analysis

Figure 4.6 depicts the theoretical times taken by each of the component when the global memory accesses are pre-staged in the shared memory. We use 3D blocking here to load the data as 3D blocks into the shared memory. We observe that although pre-staging the data in shared memory reduces a larger number of global memory loads, which result in the global memory theoretical time to reduce by almost 65%, the global memory still remains as the bottleneck of the program. As a consequence, we realized a 1.5-fold performance improvement when using shared memory over global memory.

2D vs. 3D blocking

Next, we discuss the performance impact of loading data into the shared memory as 3D blocks vs. 2D planes, or in other words, 3D blocking vs. 2D blocking. Figure 4.7 shows that changing the blocking scheme from 3D to 2D reduces the time taken by shared memory accesses. At the same time, the global memory still remains the performance bottleneck here and highlights the impact of higher memory throughput via shared memory.

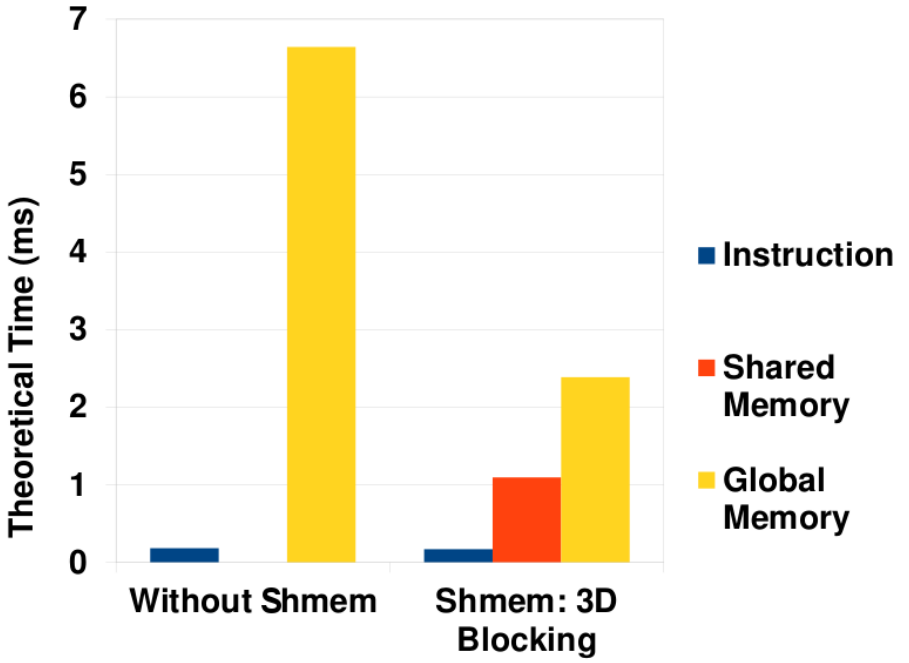


Figure 4.6: Impact of Shared Memory Usage. ('Shmem' refers to shared memory usage.)

We conclude that for 7-point stencils, as used in our FDM computation, pre-staging data in shared memory via blocking does not provide much performance improvement because the 7-point stencil computations inherently have less data reuse, and thus, cannot compensate for the added cost of loads from global memory to shared memory. Our performance model justifies and affirms our empirical results for 2D blocking and 3D blocking.

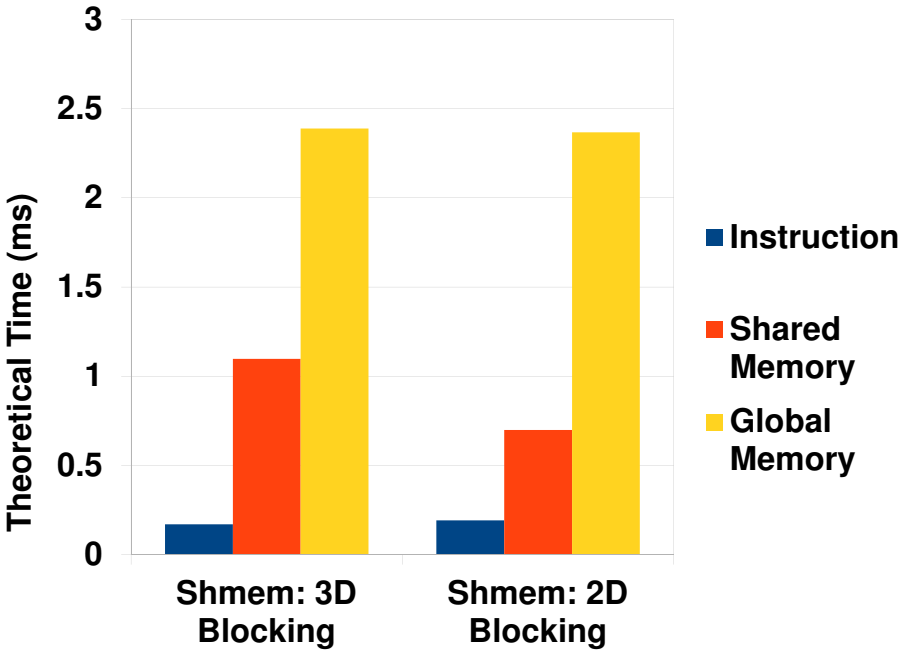


Figure 4.7: 3D vs 2D Data Blocking Technique. ('Shmem' refers to shared memory usage.)

4.4.4 Inter-node Optimizations with MPI-ACC

MPI-ACC–enabled Optimizations vs. Basic and Advanced MPI+GPU

Figure 4.8 shows the performance of the FDM-Seismology application, with nodes varying from 16 to 128, when used with and without the MPI-ACC–enabled designs. In the figure, we report the average computation time across all the nodes because the computation-communication costs vary substantially, depending on the location of the node in the structured-grid representation of FDM-Seismology. In the basic MPI+GPU case, denoted MPI+GPU for brevity, we perform both data-marshaling operations and MPI communication on the CPU. In the advanced MPI+GPU case, denoted MPI+GPU Adv, we perform the data-

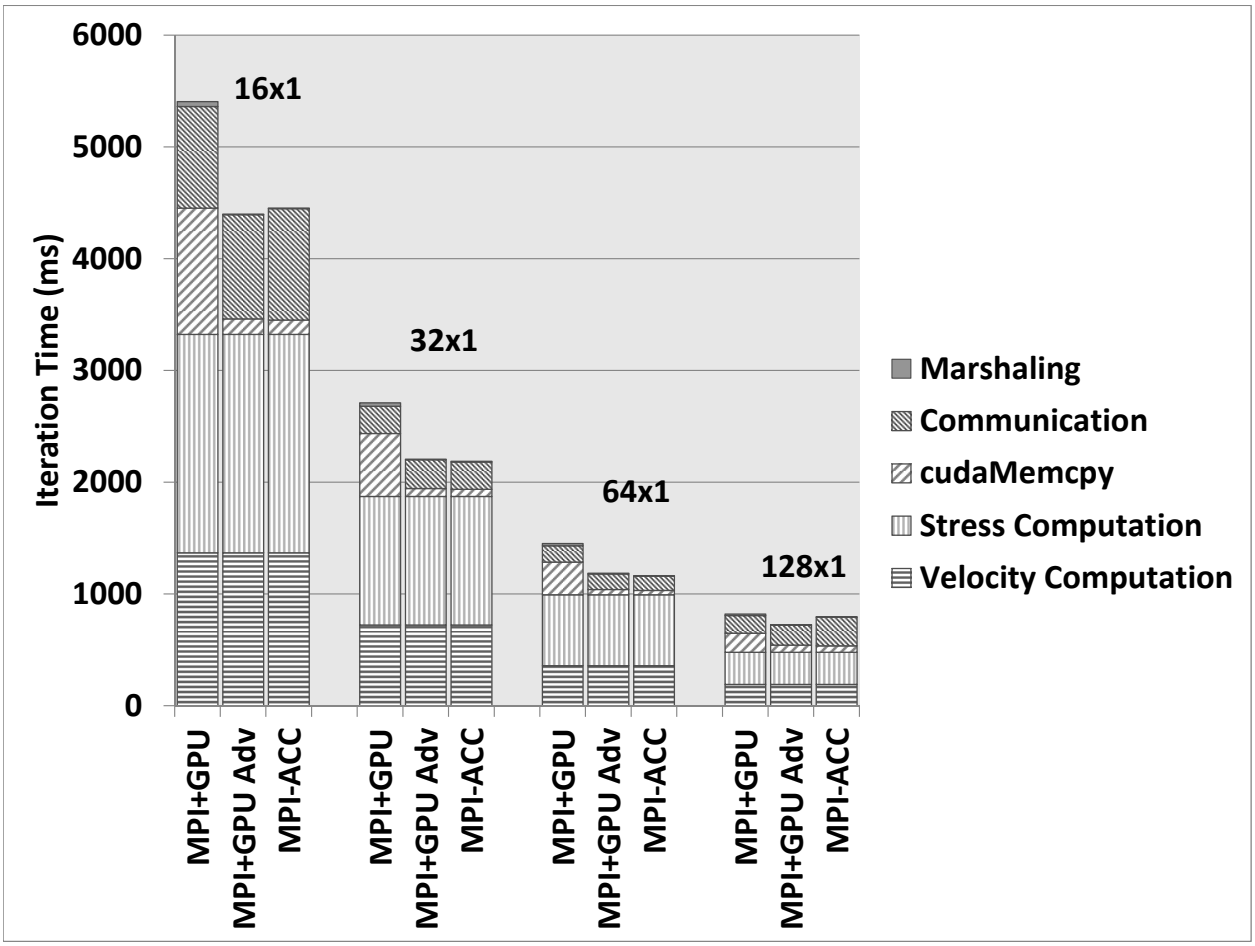


Figure 4.8: Analysis of the FDM-Seismology Application When Strongly Scaled. (The larger dataset, dataset-2, is used for these results. Note: MPI Communication represents CPU-CPU data transfer time for the MPI+GPU and MPI+GPU Adv cases and GPU-GPU (pipelined) data transfer time for the MPI-ACC case.)

marshaling operations on the GPU, while the MPI communication still takes place explicitly from the CPU.

One can see that although the velocity and stress computations take most of the application's computation time (>60%), the MPI-GPU Adv and MPI-ACC–driven design of the application see significant performance improvements over the MPI+GPU case, primarily because of the reduction in explicit bulk-data transfer operations between the CPU and

GPU, as shown in Figure 4.2c. In the MPI+GPU case, the application needs to move large wavefield data between the CPU and the GPU for the data-marshaling computation and MPI communication. On the other hand, when used with MPI-ACC, the application performs all the data-marshaling computation and MPI communication directly from the GPU, and it only needs to transfer smaller-sized wavefield data from GPU to CPU once at the end of the iteration for writing the result to the output file.

In the MPI+GPU Adv case, since the marshaling steps are performed on the GPU and much smaller-sized data arrays are moved between the CPU and GPU for MPI communication, we still benefit from avoiding the bulk data transfer steps. However, these small wavefield data movements have to be invoked many times and result in reduced programmer productivity. With MPI-ACC, the programmer enjoys productivity gain as well as the performance improvements as a result of pipelined data transfers via the CPU.

Scalability Analysis

Figure 4.9 shows the performance improvement due to the MPI-ACC–enabled GPU data-marshaling strategy over the basic hybrid MPI+GPU implementation with CPU data marshaling. We see that the performance benefits due to the GPU data-marshaling design decrease with an increasing number of nodes. The reason for this behavior is two-fold:

- For a given dataset, the per-node data size decreases as the number of nodes increases. This reduces the costly CPU-to-GPU and GPU-to-CPU bulk data transfers, as shown

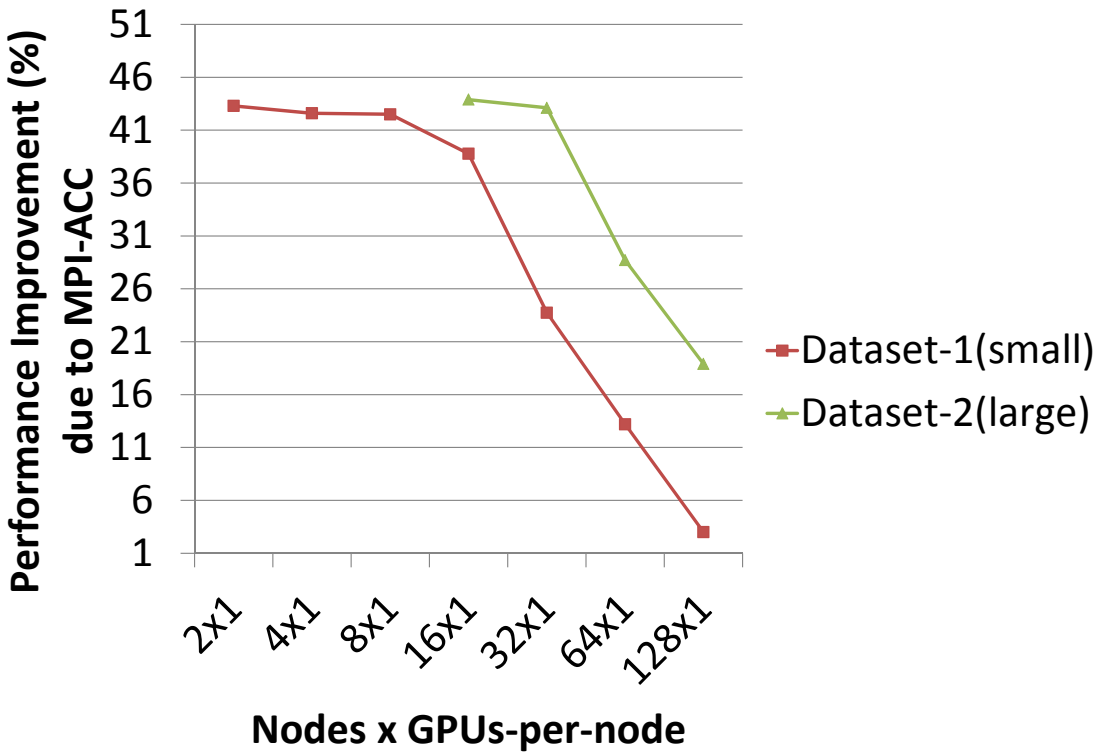


Figure 4.9: Scalability Analysis of FDM-Seismology Application with Two Datasets of Different Sizes. (The baseline for speedup is the naïve MPI+GPU programming model with CPU data marshaling.)

in Figure 4.8, and thus minimizes the overall benefits of performing data marshaling on the GPU itself.

- As the number of nodes increases, the application's MPI communication cost becomes significant when compared with the computation and data-marshaling costs. In such a scenario, the CPU-to-CPU communication of the traditional hybrid MPI+GPU implementations will have less overhead than will the pipelined GPU-to-GPU communication of the MPI-ACC–enabled design.

While the pipelined data transfer optimization within MPI-ACC improves the communication performance to a certain degree, it has negligible impact on the performance gains of the application. If newer technologies such as GPUDirect-RDMA are integrated into MPI, we can expect the GPU-to-GPU communication overhead to be reduced, but the overall benefits of GPU data marshaling itself will still be limited because of the reduced per-process working set.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this thesis, we proposed, implemented, and evaluated an online performance projection framework for best GPU device selection. The applications of our framework include runtime systems and virtual GPU environments that dynamically schedule and migrate GPU workloads in cluster environments. Our technique is based on static device profiling and online kernel characterization. To automatically obtain runtime kernel statistics with an asymptotically lower overhead, we propose a mini-emulation technique that functionally simulates a single work-group to collect per work-group statistics, which can then be used to project full-kernel statistics. Our technique is especially suitable for online performance projection for kernels with large datasets. Our experiments with GPU devices of different vendors show

our technique is able to select the best device in virtually all cases.

In this thesis, we also presented our approach to port a FDM-based seismology simulation to GPUs and how such large computations can be scaled to multiple nodes with multiple GPUs on each node. We discussed how common scientific applications, typically written in Fortran, suffer from non-ideal data layout schemes which are not amenable for GPU acceleration. We observed that the seismology application, which happens to be a 7-point 3D stencil computation, does not benefit significantly from the much faster on-chip shared memory resource of GPUs. The performance model explains this behavior and attributes this to the low data reuse due to smaller stencil point size. We also observed that for 3D stencil computations, the 2D blocking scheme is better than 3D data blocking for shared memory usage; however it does not result in any significant performance improvement. Our performance model justifies this behavior by attributing to lesser memory transactions with 2D blocking, however the global memory still persists to be the performance bottleneck which nullifies any gains from 2D blocking. We also present how large data transfer costs in typical stencil based scientific computations, can be mitigated by moving the marshaling operations to the GPU and therefore minimizing the data transfer costs. Overall, we accelerated the computation by 23-fold on a single GPU and 33-fold using dual-gpu over a single-threaded CPU performance.

We also discussed how GPU-integrated MPI libraries helps move the GPUs toward being "first-class citizens" in the hybrid clusters. Our results on HokieSpeed, a state-of-the-art CPU-GPU cluster, showed that MPI-ACC can help improve the performance of FDM-

Seismology over the default MPI+GPU implementations by enhancing the CPU-GPU and network utilization.

5.2 Future Work

We plan for various improvements in our performance projection tool. Our discussed performance projection technique can be extended to not just newer classes of GPUs such as NVIDIA Kepler architecture and AMD HSA GPUs, but also to multi-core CPUs and accelerators such as Intel Xeon Phi co-processor. We would also like our tool to have an extensible and customizable light-weight dynamic instruction generation engine, independent of third-party emulators, in order to improve the maintainability and scope of the tool and make it production quality. We plan to integrate our performance model with VOCL and study its efficacy with real-life scientific applications and compare with other run-time system such as SnuCL [23]and SOCL [18].

Bibliography

[1] TOP500, November 2013. http://www.top500.org/lists/2013/11/.

[2] *MPI: A Message-Passing Interface Standard Version 2.2.* Message Passing Interface Forum, 2009.

[3] TM Aamodt et al. GPGPU-Sim 3.x manual, 2012.

[4] A. M. Aji, M. Daga, and W. Feng. Bounding the effect of partition camping in GPU kernels. In *International Conference on Computing Frontiers (CF)*. ACM, 2011.

[5] Ashwin M. Aji, James Dinan, Darius Buntinas, Pavan Balaji, Wu-chun Feng, Keith R. Bisset, and Rajeev Thakur. MPI-ACC: An Integrated and Extensible Approach to Data Movement in Accelerator-Based Systems. In *14th IEEE International Conference on High Performance Computing and Communications*, Liverpool, UK, June 2012.

[6] AMD. APP SDK v2.7.

[7] J.P. Berenger. A perfectly matched layer for the absorption of electromagnetic waves. *Journal of Computational Physics*, 114(2):185–200, 1994.

[8] Shekhar Borkar and Andrew A Chien. The future of microprocessors. *Communications of the ACM*, 54(5), 2011.

[9] Michael Boyer, Kevin Skadron, and Westley Weimer. Automated dynamic analysis of CUDA programs. In *Proceedings of 3rd Workshop on Software Tools for MultiCore Systems*, 2010.

[10] J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2010.

[11] F. Collino and C. Tsogka. Application of the perfectly matched absorbing layer model to the linear elastodynamic problem in anisotropic heterogeneous media. *Geophysics*, 66(1):294–307, 2001.

[12] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proc. of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 2010.

[13] Kaushik Datta and Katherine A Yelick. *Auto-tuning stencil codes for cache-based multicore platforms*. PhD thesis, University of California, Berkeley, 2009.

[14] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A dynamic compiler for bulk-synchronous applications in heterogeneous systems. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, 2010.

[15] José Duato, Antonio J Pena, Federico Silla, Rafael Mayo, and ES Quintana-Orti. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *International Conference on High Performance Computing and Simulation (HPCS)*. IEEE, 2010.

[16] Wu-Chun Feng, Yong Cao, Debprakash Patnaik, and Naren Ramakrishnan. Temporal Data Mining for Neuroscience. In Wen mei W. Hwu, editor, *GPU Computing Gems*. Morgan Kaufmann, February 2011. Emerald Edition.

[17] G. Festa and S. Nielsen. PML absorbing boundaries. *Bulletin of the Seismological Society of America*, 93(2):891–903, 2003.

[18] Sylvain Henry, Denis Barthou, Alexandre Denis, Raymond Namyst, Marie-Christine Counilh, et al. Socl: An opencl implementation with automatic multi-device adaptation support. 2013.

[19] S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *International Symposium on Computer Architecture (ISCA)*. ACM, 2009.

[20] Feng Ji, Ashwin M. Aji, James Dinan, Darius Buntinas, Pavan Balaji, Rajeev Thakur, Wu-chun Feng, and Xiaosong Ma. DMA-Assisted, Intranode Communication in GPU Accelerated Systems. In *14th IEEE International Conference on High Performance Computing and Communications*, Liverpool, UK, June 2012.

[21] R.G. Joseph, G. Ravunnikutty, S. Ranka, E. D'Azevedo, and S. Klasky. Efficient GPU Implementation for Particle in Cell Algorithm. In *IEEE International Parallel Distributed Processing Symposium (IPDPS)*, pages 395–406, May 2011.

[22] Khronos Group Std. The OpenCL Specification, Version 1.0. http://www.khronos.org/registry/cl/specs/opencl-1.0.33.pdf, 2009.

[23] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. Snucl: an opencl framework for heterogeneous cpu/gpu clusters. In *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 2012.

[24] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, Ahmed Fasih, AD Sarma, D Nanongkai, G Pandurangan, P Tetali, et al. Pycuda: Gpu run-time code generation for high-performance computing. *Arxiv preprint arXiv*, 911, 2009.

[25] Dimitri Komatitsch. Fluid–solid coupling on a cluster of gpu graphics cards for seismic wave propagation. *Comptes Rendus Mécanique*, 339(2):125–135, 2011.

[26] Dimitri Komatitsch, Gordon Erlebacher, Dominik Göddeke, and David Michéa. High-order finite-element seismic wave propagation modeling with mpi on a large gpu cluster. *Journal of Computational Physics*, 229(20):7692–7714, 2010.

[27] Dimitri Komatitsch, Dominik Göddeke, Gordon Erlebacher, and David Michéa. Modeling the propagation of elastic waves using spectral elements on a cluster of 192 gpus. *Computer Science-Research and Development*, 25(1-2):75–82, 2010.

[28] B. C. Lee and D. M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[29] S. Ma and P. Liu. Modeling of the perfectly matched layer absorbing boundaries and intrinsic attenuation in explicit finite-element methods. *Bulletin of the Seismological Society of America*, 96(5):1779–1794, 2006.

[30] C. Marcinkovich and K. Olsen. On the implementation of perfectly matched layers in a three-dimensional fourth-order velocity-stress finite difference scheme. *J. Geophys. Res.*, 108(B5):2276, 2003.

[31] J. Meng, V. A. Morozov, K. Kumaran, V. Vishwanath, and T. D. Uram. GROPHECY: GPU performance projection from CPU code skeletons. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[32] J. Meng and K. Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In *International Conference on Supercomputing*. ACM, 2009.

[33] David Michéa and Dimitri Komatitsch. Accelerating a three-dimensional finite-difference wave propagation code using gpu graphics cards. *Geophysical Journal International*, 182(1):389–402, 2010.

[34] A. Nere, A. Hashmi, and M. Lipasti. Profiling Heterogeneous Multi-GPU Systems to Accelerate Cortically Inspired Learning Algorithms. In *IEEE International Parallel Distributed Processing Symposium (IPDPS),*, pages 906–920, May 2011.

[35] NVIDIA Corporation. NVIDIA Corporation. NVIDIA Visual Profiler. http://developer.nvidia.com/content/nvidiavisualprofiler.

[36] NVIDIA Corporation. NVIDIA CUDA compute unified device architecture programming guide. http://developer.download.nvidia.com/compute/cuda/ 08/NVIDIA_CUDA_Programming_Guide_0.8.pdf, 2007.

[37] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-Zee Ueng, John A. Stratton, and Wen-mei W. Hwu. Program optimization space pruning for a multithreaded GPU. In *IEEE/ACM international symposium on Code Generation and Optimization (CGO)*. ACM, 2008.

[38] Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, and Richard Vuduc. GPUPerf: A performance analysis framework for identifying potential benefits in GPGPU applications. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2012.

[39] V. Taylor, X. Wu, and R. Stevens. Prophesy: An infrastructure for performance analysis and modeling of parallel and grid applications. *SIGMETRICS Perform. Eval. Rev.*, 30(4), 2003.

[40] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2Sim: A simulation framework for CPU-GPU computing . In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[41] Vasily Volkov and James Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. In *ACM/IEEE Conference on Supercomputing (ICS)*, 2008.

[42] Hao Wang, Sreeram Potluri, Miao Luo, Ashish Singh, Sayantan Sur, and Dhabaleswar Panda. MVAPICH2-GPU: Optimized GPU to GPU communication for InfiniBand clusters. *International Supercomputing Conference (ISC) '11*, 2011.

[43] Liu Weiguo, B. Schmidt, G. Voss, and W. Muller-Wittig. Streaming Algorithms for Biological Sequence Alignment on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 18(9):1270–1281, Sept. 2007.

[44] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2010.

[45] Shucai Xiao, Pavan Balaji, James Dinan, Rajeev Thakur, Susan Coghlan, Heshan Lin, Gaojin Wen, Jue Hong, and Wu-chun Feng. Transparent accelerator migration in a virtualized GPU environment. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2012.

[46] Shucai Xiao, Pavan Balaji, Qian Zhu, Rajeev Thakur, Susan Coghlan, Heshan Lin, Gaojin Wen, Jue Hong, and Wu-chun Feng. VOCL: An optimized environment for transparent virtualization of graphics processing units. In *IEEE Innovative Parallel Computing (InPar)*, 2012.

[47] Y. Zhang and J. D. Owens. A quantitative performance analysis model for GPU architectures. In *International Symposium on High-Performance Computer Architecture (HPCA)*, 2011.