Programming High-Performance Clusters with Heterogeneous Computing Devices

Ashwin M. Aji

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

Doctor of Philosophy
in
Computer Science and Applications

Wu-chun Feng, Chair
Pavan Balaji
Keith R. Bisset
Madhav V. Marathe
Calvin J. Ribbens

January 12, 2015
Blacksburg, Virginia

Keywords: Runtime Systems, Programming Models, General Purpose Graphics Processing Units (GPGPUs), Message Passing Interface (MPI), CUDA, OpenCL

Programming High-Performance Clusters with Heterogeneous Computing Devices

Ashwin M. Aji

(ABSTRACT)

Today's high-performance computing (HPC) clusters are seeing an increase in the adoption of accelerators like GPUs, FPGAs and co-processors, leading to heterogeneity in the computation and memory subsystems. To program such systems, application developers typically employ a hybrid programming model of MPI across the compute nodes in the cluster and an accelerator-specific library (e.g.; CUDA, OpenCL, OpenMP, OpenACC) across the accelerator devices within each compute node. Such explicit management of disjointed computation and memory resources leads to reduced productivity and performance. This dissertation focuses on designing, implementing and evaluating a runtime system for HPC clusters with heterogeneous computing devices. This work also explores extending existing programming models to make use of our runtime system for easier code modernization of existing applications. Specifically, we present MPI-ACC, an extension to the popular MPI programming model and runtime system for efficient data movement and automatic task mapping across the CPUs and accelerators within a cluster, and discuss the lessons learned.

MPI-ACC's task mapping runtime subsystem performs fast and automatic device selection for a given task. MPI-ACC's data movement subsystem includes careful optimizations for end-to-end communication among CPUs and accelerators, which are seamlessly leveraged by the application developers. MPI-ACC provides a familiar, flexible and natural interface for programmers to choose the right computation or communication targets, while its runtime system achieves efficient cluster utilization.

That this work received support in part from an NVIDIA Graduate Fellowship is purely coincidental.

Acknowledgments

This dissertation is dedicated to my family, whose unconditional support gave me the motivation to pursue and complete the doctoral degree.

I am deeply grateful to each one of my committee members for their immense support and advice, ranging from technical, financial to career related matters. This dissertation would have been impossible without them.

I like to thank all my friends and colleagues within and outside Virginia Tech, who I met along the way and who helped me endure the lows and live the highs, which made my Ph.D. experience a truly memorable one.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

We begin the dissertation by motivating the chosen research problem and outlining our contributions.

1.1 Motivation

Accelerators are being increasingly adopted in today's high performance computing (HPC) clusters. A diverse set of accelerators exist, including graphics processing units (GPUs) from NVIDIA and AMD and the Xeon Phi coprocessor from Intel. In particular, GPUs have accelerated production codes for many scientific applications, including computational fluid dynamics, cosmology, and data analytics. About 15% of today's top 500 fastest supercomputers (as of November 2014) employ general-purpose accelerators [5].

Today, general-purpose heterogeneous clusters typically consist of nodes with a multi-socket multicore CPU (approximately 8 to 32 cores) along with a few GPUs (approximately 1 to 4 devices) placed on the PCIe interface. Also, these clusters predominantly have identical nodes in order to ease the burden of installation, configuration, and programming such systems; however, we also are increasingly seeing heterogeneous clusters with different nodes on the path toward 10x10 [25], 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. The *ShadowFax* cluster at the Virginia Bioinformatics Institute (VBI), Virginia Tech also consists of different sets of nodes with different CPU and accelerator configurations.

The Message Passing Interface (MPI) [6] is the de-facto standard for programming distributed memory clusters. The MPI library manages process startup, process placement, data movement and synchronization *across* nodes. Similarly, programming models such as Compute Unified Device Architecture (CUDA) [9] or Open Computing Language (OpenCL) [39] are used for kernel launches, kernel placement (device selection), memory management and synchronization among the accelerator devices *within* the node. The runtime systems for the hybrid programming model are thus fragmented, i.e., MPI across nodes and the local operating system and CUDA/OpenCL runtime systems within the node. We refer to this disjoint hybrid model as the MPI+X programming model, where MPI is used across nodes and a local "X" programming model of choice is used within each node.

Device-Task Management Each MPI process in the hybrid model consists of a host component that is run by the OS on the CPU core(s), and a device component that is run by CUDA/OpenCL on the local device(s). The host process is automatically assigned to the appropriate CPU core by the OS scheduler, and additional tools/libraries like `numactl`/`libnuma` can be used to guide the process-CPU mapping. However, the current GPU programming models require the programmers to explicitly choose the device for kernel offloading.

With increasing heterogeneity within a node, it is critical 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 on different GPU architectures. While OpenCL is a convenient platform for writing portable applications across accelerators, its performance portability remains a well-known and open issue. For instance, using the image memory subsystem and vector arithmetic can deliver up to $3\times$ performance improvement over an unoptimized OpenCL kernel on certain AMD GPUs, whereas the same optimizations may degrade the kernel's performance by about 8% on an NVIDIA GPU [30].

OpenCL has workflow abstractions called command queues through which users submit commands to a specific device. However, the OpenCL specification tightly couples a command queue with a specific single device for the entire program with no runtime support for cross-device scheduling. For best performance, the programmers thus have to find the ideal mapping of a queue to one or more devices at command queue creation time, an effort that requires a thorough understanding of the kernel characteristics, the underlying architecture, node topology, and various data-sharing costs, all of which severely hinder programmability. Researchers are exploring ways to extend the OpenCL semantics for data-parallel and task-parallel workloads by scheduling kernels across multiple OpenCL devices.

Data Movement Data movement between processes is currently limited to data residing in the host CPU memory. The ability to interact with auxiliary memory systems, such as GPU memory, has not been integrated into such data movement standards, thus providing applications with no direct mechanism to perform end-to-end data movement. Currently, transmission of data from accelerator memory must be done by explicitly copying data to host memory before performing any communication operations. This process impacts productivity and can lead to a severe loss in performance. Significant programmer effort would be required to recover this performance through vendor- and system-specific optimizations, like GPUDirect [4] and node and I/O topology awareness.

1.2 Research Questions

We identify and address three main research questions in this dissertation:

1. How to integrate auxiliary memory spaces into MPI?
2. How to incorporate dynamic device targets within MPI and X?
3. How to efficiently synchronize MPI with X?

Figure 1.1 depicts the scope of the above challenges in the context of high performance clusters.

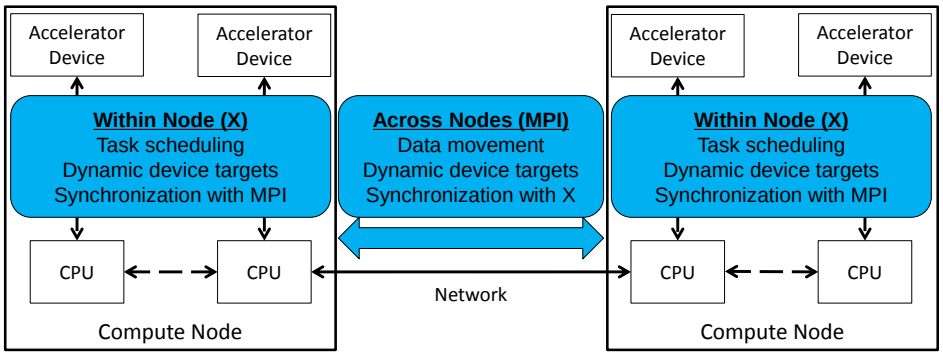


Figure 1.1: Key challenges with the MPI+X programming model

1.3 Contributions

My dissertation focuses on designing, implementing and evaluating a runtime system for heterogeneous high performance computing systems with disjoint compute and memory subsystems while extending the boundaries of current programming models. In particular, we extend the semantics of the popular MPI programming model to enable end-to-end data movement and the OpenCL and CUDA programming models for automatic task mapping among CPUs and accelerators within a cluster. The overarching contributions of my PhD dissertation are as follows:

- We design and implement MPI-ACC [15] as an extension to the popular MPI parallel programming model and a runtime system for accelerator-based heterogeneous clusters.
- We develop and evaluate a data movement subsystem that includes a wide range of optimizations for end-to-end point-to-point data movement among CPUs and accelerators, which can be seamlessly leveraged by the application developers [11, 13, 14].
- We extend the OpenCL semantics and develop a task mapping runtime subsystem that leverages a performance projection technique for fast, accurate and automatic device selection across the cluster [12, 16, 17].
- We describe the interactions between the MPI and OpenCL runtimes and discuss key driver innovations that are needed to minimize the runtime dependencies.

1.4 Lessons Learned

One of the main lessons learned was that the MPI+X programming model has programmability and performance challenges. By extending the MPI programming model to natively support GPU data structures and by extending the within-node GPU programming interface to enable automatic device management, we could write high-level scientific application code at scale. Also, the current runtime systems, i.e., the cross-node data movement and within-node task mapping runtimes, have to be codesigned to overcome current GPU driver limitations and achieve better efficiency. We encompass the inter- and intra-node runtime contributions into "MPI-ACC".

The data movement and task mapping subsystems are useful additions to the MPI runtime system. MPI-ACC provides a natural interface for programmers to specify actual devices or device abstractions as communication targets, whereas the runtime maps the communication request to the ideal device while maintaining efficient cluster utilization.

Our experiments were conducted on *HokieSpeed* – a 212 TFlop CPU-GPU cluster and on *Fire* – an eight node CPU-GPU cluster, both housed at Virginia Tech. The data movement subsystem of MPI-ACC was evaluated using microbenchmarks and applications from scientific computing domains like seismology and epidemiology. We found that the pipelined data transfers for end-to-end communication among GPUs improved the performance of MPI latency benchmarks by about 35%. We also validated via the epidemiology application that the MPI-ACC runtime system performed automatic resource management and was more scalable than the manual MPI+X programming approach. We evaluated our task scheduling system and performance projection model for best device selection by using OpenCL as the example GPU programming model and devices of different generations from both NVIDIA and AMD. Our evaluations on benchmarks and the seismology simulation showed that our model could accurately choose the best device for the given task with minimal error and low overhead.

The rest of the dissertation is organized as shown in the Figure 1.2. Related work is described in Chapter 3. In Chapter 4, we outline MPI-ACC's application programming interface (API) and semantics for its usage with GPU code; more specifically: (1) end-to-end data movement, (2) synchronization of MPI and GPU operations, and (3) dynamic device selection via a high-level device abstraction. In Chapter 5, we explain the memory management

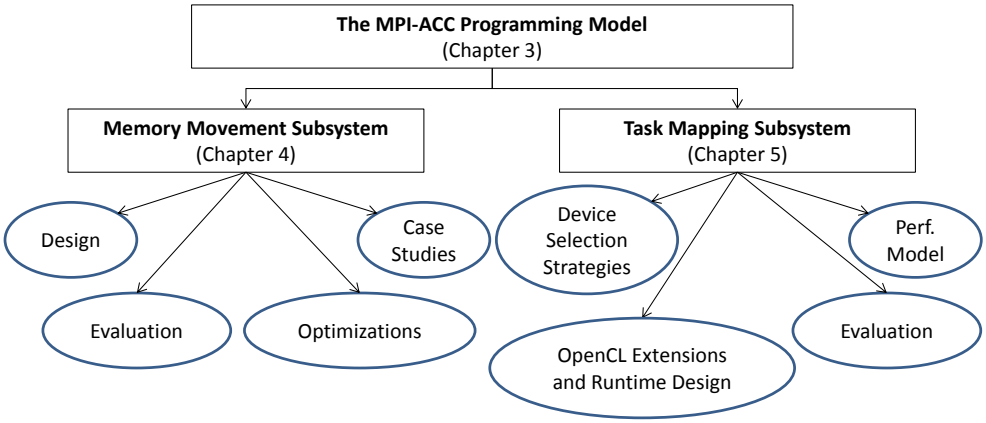


Figure 1.2: Dissertation summary

runtime subsystem of MPI-ACC, design, optimizations and evaluation with case studies. In Chapter 6, we describe and evaluate MPI-ACC's task mapping runtime subsystem and our chosen device selection strategies. In Chapter 7, we outline some of the potential future directions for this research and summarize the dissertation.

Chapter 2

Programming HPC Clusters with Heterogeneous Computing Devices

In this chapter, we outline some background information on GPU programming models, and state-of-the-art methods to write MPI+X programs for high-performance heterogeneous clusters.

2.1 Background on GPU Programming Models

Most of today's GPUs are connected to the host processor and memory through the PCIe interconnect. The high-end GPUs typically contain separate, high-throughput memory subsystems (e.g., GDDR5); and data must be explicitly moved between GPU and host memories by using special library DMA transfer operations. Some GPU libraries provide direct access to host memory, but such mechanisms still translate to implicit DMA transfers.

CUDA [9] and OpenCL [39] are two of the commonly used *explicit* GPU programming models, where GPU-specific code is written to be executed exclusively on the GPU device. CUDA is a popular, proprietary GPU programming environment developed by NVIDIA; and OpenCL is an open standard for programming a variety of accelerator plat-

forms, including GPUs, FPGAs, many-core processors, and conventional multicore CPUs. Both CUDA and OpenCL provide explicit library calls to perform DMA transfers from the host-to-device (H2D), device-to-host (D2H), device-to-device (D2D), and host-to-host (H2H). In both CUDA and OpenCL, DMA transfers involving pinned host memory provide significantly higher performance than does using pageable memory.

2.1.1 Streams and Synchronization Semantics

GPUs have hardware queues for enqueueing GPU operations; for example, NVIDIA GPUs (compute capability 2.0 and above) have one hardware queue each for enqueueing kernels, D2H data transfers, and H2D data transfers. In this way, one can potentially overlap kernel execution with H2D and D2H transfers simultaneously. In addition, CUDA and OpenCL both provide GPU workflow abstractions, called *streams* (cudaStream_t) and *command queues* (cl_command_queue).[1] A GPU stream denotes a sequence of operations that execute in issue order on the GPU [73]. Operations from different streams can execute concurrently and may be interleaved, while operations within the same stream are processed serially. Synchronization between streams is explicit, whereas the synchronization within a stream is implicit. Also, all the stream operations are asynchronous with respect to the host CPU. We note that if a data element is shared among multiple streams, say one stream for kernel execution and another for D2H transfers, the streams must be explicitly synchronized for correctness; otherwise the behavior is undefined.

2.1.2 Contexts, Platforms and Queues

In OpenCL, developers must pick one of the available platforms or OpenCL vendor implementations on the machine and create *contexts* within which to run the device code. Data can be shared only across devices within the same context. With few exceptions, typically different devices will be part of separate vendor-specific platforms; that is, a device from one vendor will not typically be part of the same platform and context as the device from another vendor. For example, the NVIDIA OpenCL implementation cannot be used to run programs on Intel Xeon Phi coprocessors and vice versa. Third-party OpenCL implementations such as SnuCL [46], however, provide a unified platform across

[1] CUDA streams and OpenCL command queues are referred to as *GPU streams* henceforth in this document.

all vendors and allow contexts to be created across devices from different vendors.

While OpenCL is a convenient platform for writing portable applications across multiple accelerators, its performance portability remains a well-known and open issue. For instance, using the image memory subsystem and vector arithmetic can bring up to $3\times$ performance improvement over an unoptimized OpenCL kernel on certain AMD GPUs, whereas the same optimizations may degrade the kernel's performance by about 8% on an NVIDIA GPU [30]. Application developers may thus want to maintain different optimizations of the same kernel for different architectures and explicitly query and schedule the kernels to be executed on the specific devices that may be available for execution.

In OpenCL, kernel objects are created per context; that is, they are shared across all devices within that context. However, the kernel launch configuration or work dimensions are set globally per kernel object at kernel launch, and per-device kernel configuration customization is possible only through custom conditional programming at the application level. No convenient mechanism exists, however, to set different kernel configurations for different kernel-device combinations dynamically. Therefore, the OpenCL interface and device scheduling are tightly coupled.

2.1.3 Data Representation and Address Spaces

Despite their apparent similarities, however, CUDA and OpenCL differ significantly in how accelerator memory is used and how data buffers are created and modified. In OpenCL, device memory allocation requires a valid *context* object. All processing and communication to this device memory allocation must also be performed by using the same context object. Thus, a device buffer in OpenCL has little meaning without information about the associated context. In contrast, context management is implicit in CUDA if the runtime library is used.

In OpenCL, data is encapsulated by a cl_mem object, whereas data is represented by a void * in CUDA. CUDA (v4.0 or later) also supports unified virtual addressing (UVA), where the host memory and all the device memory regions (of compute capability 2.0 or higher) can all be addressed by a single address space. At runtime, the programmer can use the cuPointerGetAttribute function call to query whether a given pointer refers to host or device memory. The UVA feature is currently CUDA-specific; and other accelerator models, such as OpenCL, do not support UVA.

2.2 MPI+X Hybrid Programming Models

This section describes the current ways to write MPI+X programs for high-performance heterogeneous computing devices. Since we focus on GPU computing, we use MPI+X and MPI+GPU interchangeably.

2.2.1 The Programmer's View

MPI-based applications are typically designed by identifying parallel tasks and assigning them to multiple processes. In the default MPI+GPU hybrid programming model, the compute-intensive portions of each process are offloaded to the local GPU. Data is transferred between processes by explicit messages in MPI. However, the current MPI standard assumes a CPU-centric single-memory model for communication. The default MPI+GPU programming model employs a hybrid two-staged data movement model, where MPI is used for internode communication of data residing in main memory, and CUDA or OpenCL is used within the node to transfer data between the CPU and GPU memories. Consider a simple example where the sender computes on the GPU and sends the results to the receiver GPU, which then does some more computations on the GPU. One can implement this logic in several ways using the hybrid MPI+GPU programming model as shown in Figure 2.1. In this simple set of examples, the additional host_buf buffer is used *only* to facilitate MPI communication of data stored in device memory. One can easily see that as the number of accelerators—and hence distinct memory regions per node—increases, manual data movement poses significant productivity and performance challenges.

Figure 2.1a describes the manual blocking transfer logic between host and device, which serializes GPU execution and data transfers, resulting in underutilization of the PCIe and network interconnects. Figure 2.1b shows how the data movement between the GPU and CPU can be pipelined to fully utilize the independent PCIe and network links. However, adding this level of code complexity to already complex applications is impractical and can be error prone. In addition, construction of such a sophisticated data movement scheme above the MPI runtime system incurs repeated protocol overheads and eliminates opportunities for low-level optimizations. Moreover, users who need high-performance are faced with the complexity of leveraging a multitude of platform-specific optimizations that continue

```
1  double *dev_buf, *host_buf;
2  cudaMalloc(&dev_buf, size);
3  cudaMallocHost(&host_buf, size);
4  if (my_rank == sender) { /* sender */
5    computation_on_GPU(dev_buf);
6    /* implicit GPU sync for default CUDA stream */
7    cudaMemcpy(host_buf, dev_buf, size, ...);
8    /* dev_buf is reused; async GPU kernel launch */
9    more_computation_on_GPU(dev_buf);
10   MPI_Send(host_buf, size, ...);
11 }
```

(a) Hybrid MPI+CUDA program with manual synchronous data movement (sender's logic only). This approach loses data transfer performance but gains a bit when the second GPU kernel is overlapped with MPI.

```
1  double *dev_buf, *host_buf;
2  cudaStream_t kernel_stream, streams[chunks];
3  cudaMalloc(&dev_buf, size);
4  cudaMallocHost(&host_buf, size);
5  if (my_rank == sender) { /* sender */
6    computation_on_GPU(dev_buf, kernel_stream);
7    /* explicit GPU sync between GPU streams */
8    cudaStreamSynchronize(kernel_stream);
9    for(j=0;j<chunks;j++) {
10     cudaMemcpyAsync(host_buf+offset, dev_buf+offset,
11                     D2H, streams[j], ...);
12   }
13   for(j=0;j<chunks;j++) {
14     /* explicit GPU sync before MPI */
15     cudaStreamSynchronize(streams[j]);
16     MPI_Isend(host_buf+offset, ...);
17   }
18   /* explicit MPI sync before GPU kernel */
19   MPI_Waitall();
20   more_computation_on_GPU(dev_buf);
21 }
```

(b) Hybrid MPI+CUDA program with manual asynchronous data movement (sender's logic only). This approach loses programmability but gains data transfer performance. But, the second GPU kernel is not overlapped with MPI.

Figure 2.1: Tradeoffs with hybrid MPI+CUDA program design. For MPI+OpenCL, clEnqueueReadBuffer and clEnqueueWriteBuffer would be used in place of cudaMemcpy. Similarly, clGetEventInfo or clFinish would be used in place of cudaStreamSynchronize.

to evolve with the underlying technology (e.g, GPUDirect [4]).

2.2.2 Default Synchronization Semantics

In the hybrid programming model with interleaved GPU and MPI operations, the programmer must adhere to the programming semantics of both models. When GPU data-transfer operations on GPU streams (D2H) are followed by MPI operations on the copied host data, the programmer *explicitly* waits or checks the status of the GPU streams before

calling MPI. The reason is that MPI operates on ready host buffers and, in this case, the host buffer will be undefined until the GPU stream has completed its D2H operation. Similarly, if nonblocking MPI operations are followed by GPU data transfer operations on the same data, the programmer *explicitly* waits for the completion of MPI before performing the GPU data transfer, as shown in Figure 2.1.

GPU operations are performed only on GPU data, while MPI operations are performed only on CPU data. When data changes devices (i.e., GPU data is copied to the CPU), the programming model also changes from CUDA/OpenCL to MPI. However, explicit synchronization between GPU operations and MPI is required only for dependent data operations.

Chapter 3

Related Work

We classify the related work in the literature into programming models and runtime systems, data-movement solutions and task-mapping solutions including performance models and task-distribution systems for heterogeneous computing devices in HPC clusters.

3.1 Programming Models and Runtime Systems

OpenCL [39] is an open standard and parallel programming model for programming a variety of accelerator platforms, including NVIDIA and AMD GPUs, FPGAs, the Intel Xeon Phi coprocessor, and conventional multicore CPUs. OpenCL follows a kernel-offload model, where the data-parallel, compute-intensive portions of the application are offloaded from the CPU host to the device. The hardware vendors provide their own OpenCL implementations, but they are not cross-compatible. For example, we cannot create an OpenCL context that includes devices from two different vendor platforms, which means that we cannot share memory buffers, command queues and other OpenCL data structures across the devices in the system by just using the vendor-provided OpenCL implementations.

OpenMP (v4.0) [7] and OpenACC [8] are directive-based programming models for writing parallel programs primarily

for CPUs and GPUs in single-node systems. The programmer writes a serial program with annotations to denote parallelism, heterogeneity and asynchrony. A compiler translates the annotated code to run on the multiple CPU cores or one of the available GPU devices. Furthermore, some research groups are investigating approaches to distribute the loop iterations and tasks among all the CPUs and GPUs within the node (e.g.; CoreTSAR [67]). OmpSs [28], which is an extension to the OpenMP model, also aims to run the annotated code on GPU-enabled clusters. All the above models are fork-join models where a single master thread executes sequentially until a parallel region construct is encountered. The parallel regions fork worker threads on the CPU or GPU kernels or both, then join back to the master thread to continue the serial execution.

StarPU [19] is a portable runtime system that schedules tasks across hybrid CPU-GPU architectures. The programmer implements StarPU *codelets* or task variants for the desired devices and specifies dependency constraints among them to create a task graph. StarPU schedules the tasks and manages data transfers across the different devices in the node or cluster environment. Higher-level programming models can leverage the StarPU runtime system, but the challenge lies in creating the right-sized tasks to achieve scalability and efficient resource usage.

The StarPU team have also developed SOCL [36], an OpenCL implementation that uses the StarPU runtime to schedule kernels among all the OpenCL devices in the compute environment, even if they are from different vendor platforms. SOCL provides OpenCL extensions to create command queues to scheduling contexts in addition to specific devices, which enables their scheduler to choose the ideal device for the task from all the devices within the same context. SOCL also provides OpenCL extensions for granularity adaptation of the kernels, which helps in creating the right-sized tasks for the StarPU runtime. While the command-queue extension relieves the programmer from choosing the right device for their tasks, the granularity-adaptation extension requires significant programmer effort to allow StarPU to make efficient schedules. StarPU uses historical task completion times to decide the future schedules and the relative device performances can also be provided by the application developer. Our performance modeling work for device selection does not rely on historical data and automatically computes the relative device performance. Moreover, our model is complementary to StarPU and can be used in addition to the existing ones in their scheduler.

SnuCL [46] is another cross-platform OpenCL implementation that provides the programmer with a single node view

of a cluster with OpenCL devices, where the OpenCL devices can be from multiple vendor platforms. In the SnuCL programming model, there is one host node and multiple backend nodes. SnuCL enables the application to use all the OpenCL devices in the host and backend nodes in the cluster as if they were on the host node itself. However, unlike SOCL/StarPU, the programmer has to manually choose the OpenCL device for each kernel and data transfer operation, i.e. there is no performance model for device selection.

Virtual OpenCL (VOCL) [83] and rCUDA [34] are virtual GPU models that enable the local applications to run programs on remote GPUs as if they were on the local node itself. Virtualization also enables a small set of backend GPU nodes to service multiple frontend client nodes, thereby minimizing the GPU footprint in clusters and cloud environments.

DCGN [74] is a novel programming environment that moves away from the GPU-as-a-worker programming model. DCGN assigns ranks to GPU threads in the system and allows them to communicate among each other by using MPI-like library calls. The actual control and data message transfers are handled by the underlying runtime layer, which hides the PCIe transfer details from the programmer. Our contribution is orthogonal to DCGN's in that we retain the original MPI communication and execution model while hiding the details of third-party CPU-GPU communication libraries from the end user.

All the above programming models are high level in that they use communication libraries like MPI as the underlying runtime for cluster management. While these models provide a convenient way to utilize the heterogeneous devices within a node and have shown to extend support to cluster environments, they do not show the scalability to hundreds of thousands of cores like MPI applications. We believe that the above models and runtime systems can be best used in conjunction with MPI, i.e. MPI across nodes and one of the above models within each node.

3.2 Data Movement Solutions

MVAPICH [3] is another implementation of MPI based on MPICH and is optimized for RDMA networks such as InfiniBand. From v1.8 onward, MVAPICH has included support for transferring CUDA memory regions across the

Table 3.1: Comparison of MVAPICH and Open MPI with our work.

MPI-ACC	MVAPICH / Open MPI
• Attribute-based design – Valid for both CUDA and OpenCL – No pointer check overhead • Implicit dependency resolution for mixed MPI and GPU operations – Automatic performance gain • Naturally extends to dynamic device targets, if available • Setup code – Low for P2P communication – Low for mixed MPI+GPU operations	• UVA-based design – Valid for CUDA 4.0+ and NVIDIA GPUs – Has pointer check overhead • Explicit dependency resolution for mixed MPI and GPU operations – Manual performance gain • Dynamic device targets for UVM feature of CUDA 6.0+ • Setup code – None for P2P communication – High for mixed MPI+GPU operations

network (point-to-point, collective and one-sided communications). In order to use this, however, each participating system should have an NVIDIA GPU of compute capability 2.0 or higher and CUDA v4.0 or higher, because MVAPICH leverages the UVA feature of CUDA. On the other hand, MPI-ACC takes a more portable approach: we support data transfers among CUDA, OpenCL, and CPU memory regions; and our design is independent of library version or device family. By including OpenCL support in MPI-ACC, we automatically enable data movement between a variety of devices, including GPUs from NVIDIA and AMD, IBM and Intel CPUs Intel MICs, AMD Fusion, and IBM's Cell Broadband Engine. Furthermore, we make no assumptions about the availability of key hardware features (e.g., UVA) in our interface design, thus making MPI-ACC a truly generic framework for heterogeneous CPU-GPU systems. Figure 3.1 summarizes the differences between MVAPICH and related frameworks and MPI-ACC, our contribution.

CudaMPI [49] is a library that helps improve programmer productivity when moving data between GPUs across the network. It provides a wrapper interface around the existing MPI and CUDA calls. Our contribution conforms to the MPI Standard, and our implementation removes the overhead of communication setup time, while maintaining productivity.

GPUs have been used to accelerate many HPC applications across a range of fields in recent years [56, 62, 79]. For large-scale applications that go beyond the capability of one node, manually mixing GPU data movement with MPI communication routines is still the status quo, and its optimizations usually require expertise [27, 35, 69].

3.3 Task Mapping Solutions

In this section, we describe work related to performance models and runtime systems for task-device mapping.

3.3.1 Performance Models

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

Performance Estimation

GPU performance models have been proposed to help understand the runtime behavior of GPU workloads [38, 84]. Some studies also use microbenchmarks to reveal architectural details [38, 77, 81, 84]. Although these tools provide in-depth performance insights, they often require static 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 [51, 75]. 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 [12, 26, 71]. 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 [47, 54]. Moreover, Ryoo et al. proposed additional metrics (efficiency and utilization) [65] to help prune the transformation space. Furthermore,

several autotuning techniques have been devised for specific application domains [29, 55].

The various techniques either require the source code for static 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.

Our work focuses on leveraging the familiar and popular MPI and CUDA/OpenCL programming models and make extensions to their existing semantics for accelerator-based heterogeneous clusters. We add data movement and task mapping techniques to the MPI runtime system. We use performance models for optimal task mapping, which can be leveraged by other heterogeneous runtime systems as well.

3.3.2 Runtime Schedulers

In this section, we review popular approaches for intra-application task scheduling in CPU-GPU systems and explain how our work fits into the existing literature, in terms of scheduling granularity, API design and implementation. We also discuss some inter-application schedulers and remote accelerator virtualization solutions and review their workload distribution techniques. We discuss OpenCL implementations that offer platform aggregation capabilities. Finally, we compare our design and implementation with SOCL, the work most closely related to ours.

The problem of scheduling among CPU-GPU cores is well studied and they broadly fall into two categories – intra-application scheduling and inter-application scheduling. Intra-application schedulers aim to optimize the performance of individual applications, whereas the inter-application schedulers optimize for system throughput and application turnaround time. The techniques explored by intra-application schedulers may be incorporated into the OS, accelerator runtimes or compilers, whereas the inter-application scheduling techniques may be included in cluster software stacks (GRES [50] or PBS [59]) or virtualization frameworks (VOCL [83] or rCUDA [60]). In addition, researchers have also explored optimizations for power [48] and fault tolerance [82] for the above cases. Our research deals with coarse-grained, intra-application scheduling for OpenCL programs running on CPU-GPU systems. A summary of the

Table 3.2: Literature overview of runtime schedulers for heterogeneous computing environments

Intra-application Scheduling	Data/Loop-level Parallelism	OpenMP-like: CoreTSAR [66], Kaleem [44], OmpSs [28] Custom API: Qilin [52] OpenCL-based: FluidiCL [57], Kim [45], de la Lama [32], Maestro [72]
	Task/Kernel-level Parallelism	*This dissertation*, StarPU [19] (SOCL [36])
Inter-application Scheduling	Wen [80], Ravi [63] VOCL [82, 83], rCUDA [60]	

literature related to heterogeneous runtime schedulers is shown in Table 3.2.

Intra-Application Scheduling in CPU-GPU Systems

Depending on the chosen programming model, intra-application schedulers either distribute loop iterations in directive-based applications [28, 44, 66] or distribute work groups in explicit kernel offload-based applications, i.e. *work* can mean either loop iterations or work groups. These are essentially *device aggregation* solutions, where the scheduler tries to bring homogeneity to the heterogeneous cores by giving them work proportional to their compute capabilities. Optimal work distribution techniques for both these cases can either be based on adaptive scheduling, work stealing, offline device profiling with performance modeling or machine learning techniques. We consider explicit device offload models, such as OpenCL, for this dissertation.

Scheduler Granularity The possibility of presenting multiple devices as a single device to the OpenCL layer and performing workload distribution internally has been explored in [32, 45, 52, 57, 72]. The work in [45] and [32] both follow static load partitioning approaches for intra-kernel workload distribution, but the work in [32] leverages their own API. FluidiCL [57] performs work stealing to dynamically distribute work groups among CPU-GPU cores with low overhead. Maestro [72] is a another unifying solution addressing device heterogeneity, featuring automatic data pipelining and workload balancing based on a performance model obtained from install-time benchmarking. Maestro's approach requires autotunable kernels that get the size of their workloads at runtime as parameters. Qilin [52] does adaptive mapping of computation to the CPU and GPU by using curve fitting against an evolving kernel performance

database. The above approaches provide fine grained scheduling at the kernel or loop level and exploit data parallelism in applications. In contrast, our work performs coarser grained scheduling at the *command queue* level to enable task parallelism between kernels and command queues in applications.

Application Programming Interface (API) Qilin [52] proposes a completely new API with its own compiler toolchain and [32] proposes a new API on top of OpenCL. While [45, 57] retain the existing OpenCL interface, they do not support all of the OpenCL semantics; for example, they support only synchronous kernel launches. They also do not address real application scenarios like interleaved OpenCL operations from different command queues; for example, how are concurrent kernels from different command queues implemented and how does it impact data consistency and performance? In [52, 57], it is also unclear how their approach would perform for an arbitrary number of devices. While [32] and [57] support multiple OpenCL platforms, the others require all devices to belong to a single platform. Moreover, all of their evaluation has been only on benchmarks and not on real-world applications. Furthermore, fine-grained kernel splitting performs well only for kernels with very high compute-to-memory access ratio.

We retain the full OpenCL interface and propose a few minor extensions to enable command queue scheduling. Our solution also does not enforce any restrictions on the semantics of the OpenCL program, thus allowing arbitrary interleaving of OpenCL operations. We provide a library implementation and do not rely on compiler support. Our solution works on any arbitrary number of OpenCL devices from multiple OpenCL platforms. We evaluate the efficacy of our runtime using benchmarks as well as a real world seismology simulation. The goal of our API design is to enable users to easily write task parallel OpenCL programs with multiple command queues, and let our runtime scheduler automatically handle the queue-device scheduling. The design of our framework is flexible enough to include several scheduling algorithms, and as examples, we implement a couple of schedulers in this paper. However, our framework could be extended to implement other scheduling algorithms as well.

Inter-application Scheduling

Inter-application schedulers [63, 80] distribute entire kernels from different applications across the available compute resources. Their solutions are primarily designed for multi-tenancy, power efficiency and fault tolerance in data centers. Remote accelerator virtualization solutions such as the cluster mode of SnuCL, rCUDA [60], or VOCL [83] provide seamless access to accelerators placed on remote nodes. They address the workload distribution concern in different ways. SnuCL's cluster mode permits remote accelerator access in clusters only to those nodes within the task allocation, and users have to do explicit workload distribution and scheduling. On the other hand, rCUDA enables global pools of GPUs within compute clusters, performing cross-application GPU scheduling by means of extensions to the cluster job scheduler [60]; as with SnuCL, users have to explicitly deal with load distribution and scheduling within the application. VOCL implements its own automatic scheduler, which can perform device migrations according to energy, fault tolerance, on-demand system maintenance, resource management, and load-balancing purposes [48, 82]. This scheduling mechanism, however, is limited to performing transparent context migrations among different accelerators; it is not aimed at providing performance-oriented workload distribution and scheduling.

While some of the above solutions provide scheduling support across applications, our solution provides scheduling capabilities across command queues *within* an OpenCL application.

Multiplatform OpenCL Implementations and SOCL

Some OpenCL implementations integrate multiple OpenCL platforms from different vendor implementations into a single platform, while still presenting them as separate devices. This enables cross-platform context creation and thus object sharing (buffers, events, etc.) among different compute devices from different vendors.

On the single-node mode, SnuCL limits to presenting all available OpenCL devices to applications under the same platform. The IBM OpenCL Common Runtime [40] also offers this functionality. Like the single-platform OpenCL implementations, these solutions do not provide automatic cross-device scheduling.

SOCL [36] is an OpenCL frontend to the StarPU runtime framework [19]. It provides automatic task dependency

Table 3.3: Comparison of SOCL with our work

MultiCL (Our Approach)	SOCL (with StarPU)
• Scheduling at synchronization epoch granularity • Auto-scheduling can be controlled for portions of the queue's lifetime • Kernels are tasks • Launch configuration decoupled from the launch function	• Scheduling at kernel granularity • Auto-scheduling for entire lifetime of the queue • Require specific "kernel splitter" helper functions to create tasks • Launch configuration cannot be changed

resolution and scheduling as well as performance modeling and device selection functionality. This solution applies the performance modeling at kernel granularity. In our runtime, we perform modeling at synchronization epoch granularity. Our approach enables more coarse-grained scheduling that makes device choices for task groups rather than individual tasks. Also, it speeds the model lookup time for aggregate kernel invocations, reducing runtime overhead.

In SOCL, dynamically scheduled queues are automatically distributed among devices, being bound for the entire duration of the program. In our runtime, we enable users to dynamically control the queue-device binding for specific code regions for further optimization purposes. A gist of the differences between our work and SOCL is shown in table 3.3.

In addition, our runtime enables scheduling policies both at the context level (*global*) and the command queue level (*local*). The latter may be set and reset during different phases of the program. Furthermore, our solution enables the launch configuration to be decoupled from the launch function, providing kernel-device configuration customization capabilities. A typical use case is to set different kernel launch configurations for different device types depending on the hardware and occupancy limitations.

Chapter 4

The MPI-ACC Programming Model

In this chapter, we discuss the programming semantics of the default MPI+GPU model and MPI-ACC as well as explain the design tradeoffs that were considered for MPI-ACC. We also describe the OpenCL extensions that we propose to enable custom scheduling in GPU programs. This chapter is based on [11] and [17].

4.1 Designing the MPI-ACC Programming Model for Data Communication

4.1.1 Motivation

To bridge the gap between the disjointed MPI and GPU programming models, GPU-integrated MPI solutions are being developed, such as our MPI-ACC [13] framework and MVAPICH-GPU [78] 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 addition to enhanced programmability, transparent architecture-specific and vendor-specific performance optimizations can be provided within the MPI layer.

With MPI-ACC, programmers only need to write GPU kernels and regular host CPU codes for computation and invoke the standard MPI functions for CPU-GPU data communication, without worrying about the aforementioned complex data-movement optimizations of the diverse accelerator technologies. The task-mapping subsystem of MPI-ACC aims to automatically choose the ideal device for the given compute kernel.

4.1.2 The Application Programming Interface (API) Design

While, conceptually, most GPU-integrated MPI frameworks are broadly similar to each other, they differ with respect to the user programming semantics they provide. Specifically, how GPU execution and communication integrates with MPI communication can be different for different frameworks. In this section, we discuss two such programming semantics that are found in current GPU-integrated MPI frameworks: (1) UVA-based design and (2) MPI attributes–based design.

UVA-based design

UVA is a CUDA-specific concept that allows information about the buffer type to be encoded inside a void * size argument. This allows both the host memory and the GPU memory to be represented within a common 64-bit virtual address space. In such a model, the user would pass a void * communication buffer argument to MPI, as it would do in a traditional MPI library (Figure 4.1a). The MPI implementation would, internally, query for the buffer type attribute using CUDA's cuPointerGetAttribute function. With this, the MPI implementation can identify whether the buffer resides on host memory or on GPU memory and thus decide whether to perform a pipelined data transfer for GPU data over the PCIe and network interconnects or to fall back to the traditional CPU data-transfer logic. The cuPointerGetAttribute function can also be used to query the actual GPU device number on which the buffer resides.

```
1  double *dev_buf, *host_buf;
2  if (my_rank == sender) { /* send from GPU (CUDA) */
3    MPI_Send(dev_buf, ...);
4  } else { /* receive into host */
5    MPI_Recv(host_buf, ...);
6  }
```

(a) UVA-based design: example MPI code where a device buffer is sent and received as a host buffer.

```
1  double *cuda_dev_buf; cl_mem ocl_dev_buf;
2  /* initialize a custom type */
3  MPI_Type_dup(MPI_CHAR, &type);
4  if (my_rank == sender) { /* send from GPU (CUDA) */
5    MPI_Type_set_attr(type, BUF_TYPE, BUF_TYPE_CUDA);
6    MPI_Send(cuda_dev_buf, type, ...);
7  } else { /* receive into GPU (OpenCL) */
8    MPI_Type_set_attr(type, BUF_TYPE,
9                      BUF_TYPE_OPENCL);
10   MPI_Recv(ocl_dev_buf, type, ...);
11 }
12 MPI_Type_free(&type);
```

(b) MPI Attribute-based design: example MPI code where a device CUDA buffer is sent and received as an OpenCL device buffer.

Figure 4.1: Design of GPU-integrated MPI frameworks.

The MVAPICH2-GPU implementation [78], for instance, uses the UVA model. However, the cuPointerGetAttribute function is expensive relative to extremely low-latency communication times and can add significant overhead to host-to-host communication operations. Figure 4.2 shows the impact of this query on the latency of intra-node, *CPU-to-CPU*, communication using MVAPICH v1.8 on the HokieSpeed CPU-GPU cluster.

Apart from the obvious downside that UVA is CUDA-specific and is not relevant to other programming models such as OpenCL, this approach is fundamentally limited by the amount of information that can be passed by the user to MPI.

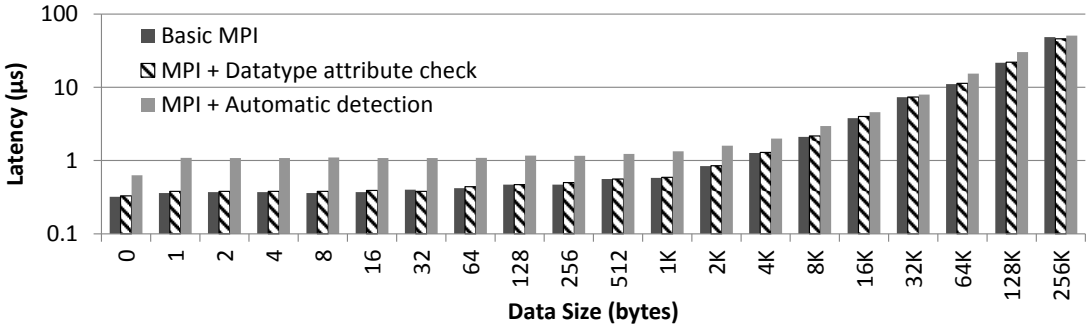


Figure 4.2: Overhead of runtime checks incurred by intranode *CPU-CPU* communication operations. The slowdown from automatic detection (via cuPointerGetAttribute) is 23% to 235%, while the slowdown from the attribute check is at most only 3%.

Specifically, only information that can be encoded within a 64-bit space can be passed to the MPI implementation. For example, the user cannot inform the MPI implementation which stream (in CUDA) or command queue (in OpenCL) can be used for the data transfer or whether MPI needs to order the data transfer relative to an event on a stream or command queue.

MPI Attribute–based Design

Another programming semantic that can be exposed by the MPI implementation is to use MPI communicator or datatype *attributes*. MPI communicators specify a collection of MPI processes and a communication context. MPI datatypes are used to specify the type and layout of buffers passed to the MPI library. The MPI standard defines an interface for attaching metadata to MPI communicators or datatypes through attributes. These attributes can be used to indicate buffer type and any other information to the MPI library (Figure 4.1b). Communicator attributes allow processes to mark a communicator as "special" in that it would always move data to/from GPUs. However, they are restricted in that they cannot move data from the GPU on one process to host memory of another process. Datatype attributes allow processes to perform a specific communication to/from GPU memory or host memory and thus are fully generic and can describe any CPU/GPU communication pattern desired by the user. This approach introduces a lightweight runtime attribute check to each MPI operation, but the overhead is negligible, as shown in Figure 4.2. Moreover, this approach is more extensible and maintains compatibility with the MPI standard. Our MPI-ACC framework uses this communication model.

An important capability of this model is that there is no restriction on the amount of information that the user can pass to the MPI implementation. The MPI implementation can define attributes for the buffer type, buffer locality (e.g., which GPU), which stream to use, ordering semantics, or basically anything else. The MPI implementation can allow some of these attributes to be optional for the user, for programming convenience, without restricting the user from providing it as needed.

4.1.3 Discussion on the Synchronization Semantics

In this section, we describe the synchronization and ordering semantics when GPU execution and data movement operations are interleaved with GPU-integrated MPI libraries [11].

UVA-Based Design

In the naïve MPI+GPU programming models, the synchronization semantics occurred only when the data changed devices; that is, if GPU data was copied to the CPU, the programming semantics would also change from implicit synchronization in CUDA/OpenCL to explicit synchronization in MPI or vice versa. However, since GPU-integrated MPI can operate directly on GPU data, the synchronization semantics must be carefully defined. If GPU operations on a CUDA stream, such as CUDA kernels or host-to-device (H2D) transfers, are followed by direct MPI operations on the same data (i.e., if there is data dependence), there should be some form of synchronization (explicit or implicit) between the GPU operation and the MPI call even though the data has not changed devices. If the MPI call does not have any data dependence with the preceding GPU operation, then synchronization is not needed.

In the GPU programming model, however, streams are used to implicitly indicate data dependence and maintain execution-order semantics. We could imagine MPI's data-dependent operations as GPU operations that belong to the same stream as the preceding GPU operation, but it is impossible to pass additional information, such as the dependent GPU stream, to the UVA-based MPI model by just a `void *` argument. Since the UVA-based model has no mechanism to implicitly express data dependence, GPU operations and their dependent MPI function calls are required to *explicitly* synchronize for correctness. On the other hand, if a GPU operation follows a nonblocking MPI operation, MPI semantics requires that MPI be *explicitly* waited upon before reusing the data.

Consider a synthetic MPI application example with N processes, where MPI process 0 computes a large array on its local GPU and sends chunks of the array directly from the GPU to the remaining $N-1$ processes, similar to a scatter operation. For the sake of argument, let us also assume that the GPU array computation is data parallel, that is, the array elements can be processed independently and can be pipelined. This example application can be implemented by

```
1  cudaStream_t myStream[N];
2  for(rank = 1; rank < N; rank++) {
3   fooKernel<<<b, t, myStream[rank]>>>
4                           (dev_buf+offset);
5  }
6  for(rank = 1; rank < N; rank++) {
7   /* explicit GPU stream sync before MPI */
8   cudaStreamSynchronize(myStream[rank]);
9   MPI_Send(dev_buf+offset, rank, ...);
10 }
```

(a) Simple UVA-based design: explicit GPU synchronization with synchronous MPI.

```
1  cudaStream_t myStream[N];
2  int processed[N] = {1, 0};
3  for(rank = 1; rank < N; rank++) {
4   fooKernel<<<b, t, myStream[rank]>>>(dev_buf+offset);
5  }
6  numProcessed = 0; rank = 1;
7  while(numProcessed < N - 1) {
8   /* explicit GPU stream query before MPI */
9   if (cudaStreamQuery(myStream[rank])==cudaSuccess) {
10     MPI_Isend(dev_buf+offset, rank, ...);
11     numProcessed++;
12     processed[rank] = 1;
13  }
14  MPI_Testany(...); /* check progress */
15  flag = 1;
16  if(numProcessed < N - 1) /* find next rank */
17    while(flag) {
18     rank=(rank+1)%N; flag=processed[rank];
19    }
20 }
21 MPI_Waitall();
```

(b) Advanced UVA-based design: explicit GPU synchronization with asynchronous MPI.

Figure 4.3: Data-dependent MPI+GPU program with explicit GPU synchronization and designed with UVA-based MPI.

using the UVA-based design in a couple of ways as shown in Figure 4.3, where the GPU kernel execution is pipelined with data movement, but MPI can be synchronous or asynchronous. Clearly, while the example in Figure 4.3a is simpler, it is also less efficient because of the blocking GPU and MPI calls. Moreover, while the GPU kernels in the different streams can finish in any order, the program waits for them in issue order. This unnecessary wait is removed in Figure 4.3b, but the code becomes more complex.

Disadvantages of the UVA-Based Design While the UVA-based design provides an ideal API by perfectly conforming to the MPI standard for end-to-end CPU-GPU communication, its code semantics forces explicit synchronization between data-dependent (ordered) and interleaved GPU and MPI operations. The MPI implementation can potentially avoid the explicit synchronization semantics by conservatively invoking cudaDeviceSynchronize before performing

data movement, but this approach will obviously hurt performance and is impractical. Moreover, the UVA-based design is not extensible for other accelerator models such as OpenCL that do not support UVA, because it is impossible to pass the cl_context and cl_command_queue arguments to MPI through just the void * argument.

MPI Attribute–based Design

Since MPI can directly operate on GPU data, the synchronization semantics of the MPI attribute-based model must also be carefully defined, that is, explicit vs. implicit. Of course, one can treat the attribute-based model just like the UVA-based model and introduce explicit synchronization semantics between data-dependent GPU and MPI calls; but we can do better with this model because there is no restriction in the amount of information that the user can pass to the MPI implementation via the MPI attribute metadata. Since GPU streams implicitly indicate data dependence on GPUs, the programmer can now pass the stream parameter itself as one of the MPI attributes. The MPI implementation can use this stream information to perform additional optimizations for best performance. For example, if a stream parameter is associated with an asynchronous MPI_Isend call, the stream could be added to a stream pool that is periodically queried for completion instead of blocking on cudaStreamSynchronize immediately. The MPI implementer is free to apply different heuristics and additional optimizations on the stream parameter, as needed. Since there exists a mechanism in the attribute-based model to implicitly express data dependence, GPU operations and their dependent MPI function calls can be either implicitly or explicitly synchronized for correctness. On the other hand, if a GPU operation follows a nonblocking MPI operation, MPI semantics requires that MPI be explicitly waited upon before reusing the data.

The synthetic MPI application example from Figure 4.3 can now be easily implemented in MPI-ACC by using the attribute-based design and implicit synchronization as shown in Figure 4.4. Note that the stream parameter that is passed to MPI_Isend is different for each loop iteration (or rank) and we have to set that attribute in every loop. The GPU buffer attribute type is constant and is set once before the loop execution. This example is fully nonblocking with asynchronous and interleaved GPU and MPI operations. With this design, we can do away with the complex logic of looping over the GPU streams and explicitly coordinating between the GPU and MPI operations. Instead, by passing

```
1  cudaStream_t myStream[N];
2  for(rank = 1; rank < N; rank++) {
3   fooKernel<<<b, t, myStream[rank]>>>(dev_buf+offset);
4   /* implicit GPU stream sync before MPI */
5   MPI_Type_dup(MPI_CHAR, &new_type);
6   MPI_Type_set_attr(new_type, BUF_TYPE, BUF_TYPE_CUDA);
7   MPI_Type_set_attr(new_type, STREAM_TYPE, myStream[rank]);
8   MPI_Isend(dev_buf+offset, new_type, rank, ...);
9   MPI_Type_free(&new_type);
10 }
11 /* explicit MPI sync */
12 MPI_Waitall();
```

Figure 4.4: Data-dependent MPI+GPU program designed with the MPI attribute-based design of MPI-ACC. The example showcases *implicit* GPU synchronization with asynchronous MPI.

more information to MPI, the synchronization semantics can be implicitly controlled within the MPI implementation.

Design of Stream Synchronization in MPI-ACC Figure 4.5 illustrates our approach to synchronizing GPU and MPI operations within MPI-ACC. We consider only GPU-specific MPI operations for this discussion; the CPU data is handled separately as before. When the programmer initiates an asynchronous MPI call on a GPU buffer with a dependent stream parameter, we have two options: (1) we can use the application's stream itself for the data transfers, which means that we avoid the complex management of streams and their synchronization within MPI, or (2) we can wait for the completion of the application stream and use MPI-ACC's multiple internal streams for more efficient data transfer. We use the second approach because, in practice, multiple streams are more efficient for pipelining, although their management can become complex. Once the user initiates an MPI command, we simply create a corresponding request object in MPI-ACC, add it to the outstanding GPU request pool, and return. We do not query immediately or wait for the stream object. MPICH's progress engine periodically checks for unfinished MPI transactions and tries to complete them when possible. We leverage and extend the progress engine to periodically query for all the unfinished stream requests. If we find that a stream request has completed, it means that the data dependence semantics have been observed and that we are free to communicate the corresponding GPU data by using our internal streams. For every completed stream request, we follow MPI's communication protocol to send the rest of the data to the receiver, namely, send the Ready-To-Send (RTS) packet to receiver, wait for the Clear-To-Send (CTS) packet, and then transfer the actual GPU payload data to the receiver. On the other hand, if the programmer initiates an MPI call without a dependent stream parameter, it means that the data is immediately ready to be transferred. In this case, we do not add

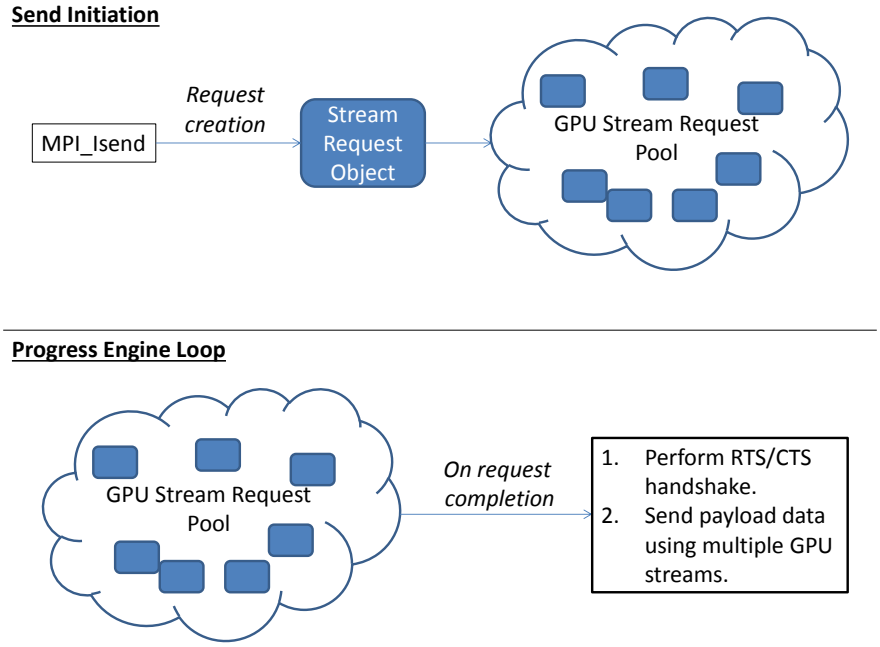


Figure 4.5: MPI-ACC's design for MPI+GPU synchronization. Example: MPI_Isend.

this request to the stream pool but directly send an RTS packet to the receiver to initiate data communication.

4.2 Extending the OpenCL Programming Model for Task Scheduling

In this section, we describe our proposed minor OpenCL API extensions to express global and local scheduling policies and the proposed OpenCL functions for decoupling kernel launches from the actual device. Our proposed extensions enable programmers to perform algorithmic decompositions among the queues and let the specified scheduler map them to the underlying devices. Table 4.1 summarizes our proposed extensions.

4.2.1 Global Scheduling at the Context

We extend the OpenCL context to specify a context-wide global queue scheduling policy. The global scheduling policy influences the final queue-device mapping. On the other hand, the queue-specific local scheduling options will determine if a specific command queue is participating in automatic scheduling or the code extent through which the

Table 4.1: Proposed extensions to the OpenCL specification

CL Function	CL Extensions	Parameter Names	Options
clCreateContext	New parameters and options	CL_CONTEXT_SCHEDULER	ROUND_ROBIN AUTO_FIT
clCreateCommandQueue	New parameters	SCHED_OFF SCHED_AUTO_STATIC SCHED_AUTO_DYNAMIC SCHED_KERNEL_EPOCH SCHED_EXPLICIT_REGION SCHED_ITERATIVE SCHED_COMPUTE_BOUND SCHED_IO_BOUND SCHED_MEMORY_BOUND	N/A
clSetCommandQueueSchedProperty	New CL API		
clSetKenelWorkGroupInfo	New CL API	N/A	N/A

queue is eligible for automatic scheduling.

We propose a new parameter to the context property called CL_CONTEXT_SCHEDULER to express the global scheduling policies. Currently, we support the following global scheduler policies: round robin and autofit. The round-robin policy schedules the command queue to the next available device when the scheduler is triggered. This approach is expected to cause the least overhead but may not always produce the optimal queue-device map. On the other hand, the autofit policy decides the most optimal queue-device mapping when the scheduler is triggered. The global policies, in conjunction with the local command queue–specific options, will determine the final queue-device mapping.

4.2.2 Local Scheduling Options at the Command Queue

While command queues that are created with the same context share data and kernel objects, they also share the context's global scheduling policy. We extend the OpenCL command queue to specify a local scheduling option that is queue-specific. The combination of global-local scheduler policies can be leveraged by the runtime to result in a more optimal device mapping. The command queue properties are implemented as bitfields, so the user can specify a combination of local policies.

Setting the queue property to either SCHED_AUTO_$\star$ or SCHED_OFF determines whether the particular queue is opting in or out of the automatic scheduling, respectively. For example, an intermediate or advanced user may want to manually optimize the scheduling of just a subset of the available queues by applying the SCHED_OFF flag to them,

while the remaining queues may use the SCHED_AUTO_DYNAMIC flag to participate in automatic scheduling. Static vs. dynamic automatic scheduling provides a tradeoff between speed and optimality. Command queue properties can also specify scheduler triggers to control the scheduling chunk size, frequency of scheduling, and scheduling code regions. For example, the SCHED_KERNEL_EPOCH flag denotes that scheduling should be triggered after a batch of kernels (*kernel epoch*) are synchronized and not after individual kernels. The SCHED_EXPLICIT_REGION flag denotes that scheduling for the given queue should be triggered between explicit start and end regions in the program, and the new clSetCommandQueueSchedProperty OpenCL command is used to mark the scheduler region and set more scheduler flags if needed. Queue properties may also be used to provide optimization hints to the scheduler. Depending on the expected type of computation in the given queue, the following properties may be used: SCHED_COMPUTE_BOUND, SCHED_MEM_BOUND, SCHED_IO_BOUND, or SCHED_ITERATIVE. For example, if the SCHED_COMPUTE_BOUND flag is used, the runtime chooses to perform minikernel profiling to reduce overhead.

4.2.3 Specifying Device-Specific Kernel Options

The parameters to the OpenCL kernel launch functions include a command queue, a kernel object, and the kernel's launch configuration. The launch configuration is often determined by the target device type, and it depends on the device architecture. Currently, per-device kernel configuration customization is possible only through custom conditional programming at the application level. The device-specific launch function forces the programmer to manually schedule kernels on a device, which leads to poor programmability.

We propose a new OpenCL API function called clSetKernelWorkGroupInfo to independently set unique kernel configurations to different devices. Its signature is shown in Figure 4.6. The purpose of this function is to enable the programmer to separately express the different combinations of kernel configuration and devices beforehand so that when the runtime scheduler maps the command queues to the devices, it can also profile the kernels using the device-specific configuration that was set before. The clSetKernelWorkGroupInfo function may be invoked at any time before the actual kernel launch. If the launch configuration is already set before the launch for each device, the runtime simply uses the device-specific launch configuration to run the kernel on the dynamically chosen

```
1 cl_int clSetKernelWorkGroupInfo(cl_kernel  kernel,
2     cl_device_id  *devices,
3     cl_uint num_devices,
4     cl_uint work_dim,
5     const size_t *global_work_offset,
6     const size_t *global_work_size,
7     const size_t *local_work_size )
```

Figure 4.6: API proposal to decouple kernel launch configuration with the actual launch itself to give more scheduling options to the runtime.

device. We do not change the parameters to the `clEnqueueNDRangeKernel` and other launch API, but the kernel configuration parameters are ignored if they are already set by using `clSetKernelWorkGroupInfo`.

4.3 Codesigning the Data Movement and Task Scheduler API

While the data movement library ideally interfaces with the high-level device abstractions provided by the GPU runtime, sometimes, it may be necessary to implement some device-specific optimizations. For example, pinned memory for device-host RDMA cannot be shared across devices, and so the MPI runtime may want to query for the queue's device use device-specific resources. Furthermore, the task scheduler runtime may change the queue-device association whenever the scheduler gets triggered. While the `clGetCommandQueueInfo` OpenCL API supports the `CL_QUEUE_DEVICE` parameter to return the device that was used during queue creation, we add a new flag `CL_QUEUE_LATEST_DEVICE` property to the command queue to retrieve the latest device that was scheduled for the command queue. For example, the MPI runtime can query for the latest device that was associated with a queue after a data read/write and then choose the respective buffer pool for pipelining.

4.4 Conclusion

By using three implementations of a synthetic example program (Figures 4.3 and 4.4), we demonstrated that with the UVA-based design one can use only the explicit synchronization method. On the other hand, with the MPI attribute-based design, we can use either explicit or implicit synchronization, and the programmer can choose the preferred

programming style. The attribute-based design can be considered somewhat like a superset of the UVA-based design. To use implicit GPU synchronization with attribute-based design, the programmer sets the stream parameter as an MPI attribute, while explicit synchronization can be used by simply not setting it. The explicit synchronization of the advanced UVA approach (Figure 4.3b) is more complex to code when compared with the simple UVA- and attribute-based approaches but is more likely to achieve the best performance. On the other hand, the attribute-based implicit GPU synchronization example is the most straightforward to code, but its performance depends on MPI's internal implementation, for example, the stream request pool management. We have chosen the attribute-based design for MPI-ACC.

To address the problem of static device management and scheduling in OpenCL, we proposed to add global and local scheduling policies to the OpenCL specification. These policies can be leveraged by a runtime system to schedule command queues and kernels among the available devices, as we discuss in Chapter 6. Our proposed scheduling policies are specified via new attributes to the `cl_context` and `cl_command_queue` objects. Our hierarchical scheduling policies provide sufficient information to a runtime scheduler to perform ideal device mapping, thereby enabling the average user to focus on the algorithm-level parallelism rather than scheduling. Attributes may be applied for the entire lifetime of the command queues, implicit synchronization epochs, or any explicit code regions. We also proposed and defined a new OpenCL function to separately specify per-device kernel execution configurations, which enables the scheduler to choose the appropriate configuration at kernel launch time, thereby associating a kernel launch with a high level command queue rather than a low level physical device. The advanced users can choose to not use any of the above scheduling policies and functions in order to manually schedule and extract performance, as usual.

Chapter 5

Data Movement with MPI-ACC

In this chapter, we discuss the data-movement subsystem of MPI-ACC. We discuss the optimizations for end-to-end data transfers among accelerators and evaluate them using microbenchmarks and case studies from epidemiology and seismology modeling. This chapter is based on [13, 14] and focuses on inter-node communication optimizations. Our related work on intra-node data movement optimizations [41, 42] is not included in this chapter.

5.1 The MPICH Software Stack

MPICH [2] is an open-source, high-performance MPI implementation. We extend MPICH to implement MPI-ACC's accelerator communication support. MPICH and its derivatives form the most widely used implementations of MPI in the world. They are used exclusively on nine of the top 10 supercomputers (November 2014 ranking), including the world's current fastest supercomputer: Tianhe-2.

The communication software stack is shown in Figure 5.1. The application invokes the MPI library API, which in turn calls a device layer abstraction. The 'CH3' device is the default example implementation of the MPICH ADI3 (Abstract Device Interface) that provides an implementation of the ADI3 using a relatively small number of functions.

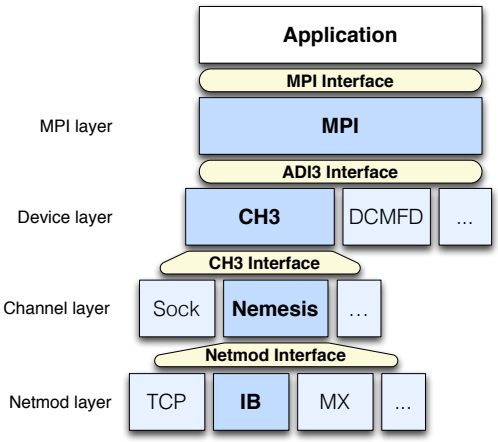


Figure 5.1: The MPICH software stack.

The device implements communication channels. A channel provides routines to send data between two MPI processes and to make progress on communications. A channel may define additional capabilities (optional features) that the CH3 device may use to provide better performance or additional functionality. Different channels may be selected at build time, and we choose the 'Nemesis' default communication channels in our implementation. Multiple network module, or *netmod*, implementations are available within Nemesis, out of which we implement MPI-ACC within the InfiniBand netmod.

The MPICH design principle is based on the one-size-does-not-fit-all policy. The MPICH design is modular and broad enough to support as many architectures and communication interfaces as possible, while performing key optimizations within each module at any layer in the software stack. Third-party vendors like MVAPICH, Cray and IBM may extend MPICH to suit their specific architectural needs while enjoying the portability of MPI. MPI-ACC is similarly extended by optimizing key components in the CH3, Nemesis and InfiniBand (IB) layers of the communication stack.

5.2 Optimizations

Once MPI-ACC has identified a device buffer, it leverages PCIe and network link parallelism to optimize the data transfer via pipelining. Pipelined data transfer parameters are dynamically selected based on NUMA and PCIe affinity to further improve communication performance. We discuss the pipelining and OpenCL metadata optimizations in

this chapter, but the other optimizations can be found in [13].

5.2.1 Data Pipelining

We hide the PCIe latency between the CPU and GPU by dividing the data into smaller chunks and performing pipelined data transfers between the GPU, the CPU, and the network. To orchestrate the pipelined data movement, we create a temporary pool of host-side buffers that are registered with the GPU driver (CUDA or OpenCL) for faster DMA transfers. The buffer pool is created at MPI_Init time and destroyed during MPI_Finalize. The system administrator can choose to enable CUDA and/or OpenCL when configuring the MPICH installation. Depending on the choice of the GPU library, the buffer pool is created by calling either cudaMallocHost for CUDA or clCreateBuffer (with the CL_MEM_ALLOC_HOST_PTR flag) for OpenCL.

To calculate the ideal pipeline packet size, we first individually measure the network and PCIe bandwidths at different data sizes (Figure 5.2), then choose the packet size at the intersection point of the above channel rates, 64 KB for our experimental cluster (section 5.4). If the performance at the intersection point is still latency bound for both data channels (network and PCIe), then we pick the pipeline packet size to be the size of the smallest packet at which the slower data channel reaches peak bandwidth. The end-to-end data transfer will then also work at the net peak bandwidth of the slower data channel. Also, only two packets are needed to do pipelining by double buffering: one channel receives the GPU packet to the host while the other sends the previous GPU packet over the network. We therefore use two CUDA streams and two OpenCL command queues per device per MPI request to facilitate pipelining.

The basic pipeline loop for a "send" operation is as follows ("receive" works the same way, but the direction of the operations is reversed). Before sending a packet over the network, we check for the completion of the previous GPU-to-CPU transfer by calling cudaStreamSynchronize or a loop of cudaStreamQuery for CUDA (or the corresponding OpenCL calls). However, we find that the GPU synchronization/query calls on *already completed* CPU-GPU copies cause a significant overhead in our experimental cluster, which hurts the effective network bandwidth and forces us to

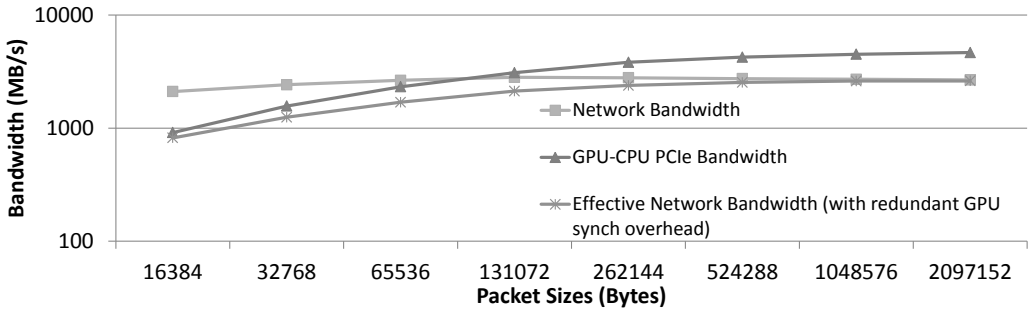


Figure 5.2: Choosing the pipeline parameters: network – InfiniBand, transfer protocol – R3.

choose a different pipeline packet size. For example, we measure the cost of stream query/synchronization operations as approximately 20 μs, even though the data transfer has been completed. Moreover, this overhead occurs every time a packet is sent over the network, as shown in Figure 5.2 by the "Effective Network Bandwidth" line. We observe that the impact of the synchronization overhead is huge for smaller packet sizes but becomes negligible for larger packet sizes (2 MB). Also, we find no overlap between the PCIe bandwidth and the effective network bandwidth rates, and the PCIe is always faster for all packet sizes. Thus, we pick the *smallest* packet size that can achieve the peak effective network bandwidth (in our case, this is 256 KB) as the pipeline transfer size for MPI-ACC. Smaller packet sizes ($<$256 KB) cause the effective network bandwidth to be latency-bound and are thus not chosen as the pipeline parameters. In MPI-ACC, we use the pipelining approach to transfer large messages—namely, messages that are at least as large as the chosen packet size—and fall back to the nonpipelined approach when transferring smaller messages.

5.2.2 Dynamic Choice of Pipeline Parameters

The effective data transfer bandwidth out of a node in current heterogeneous clusters may vary significantly. A common scenario of heterogeneous performance involves multisocket, multicore CPU architectures that may have multiple memory and PCIe controllers (GPU or IB) per socket, where the access latencies vary significantly depending on the affinity of the executing CPU core and the target memory or PCIe controller. The sockets within a die are connected via quick links, such as QPI (Intel) and HT (AMD), and the additional intersocket data transfers hurt performance. For example, if the target memory module controller and the GPU PCIe controller are on different sockets, the GPU-CPU

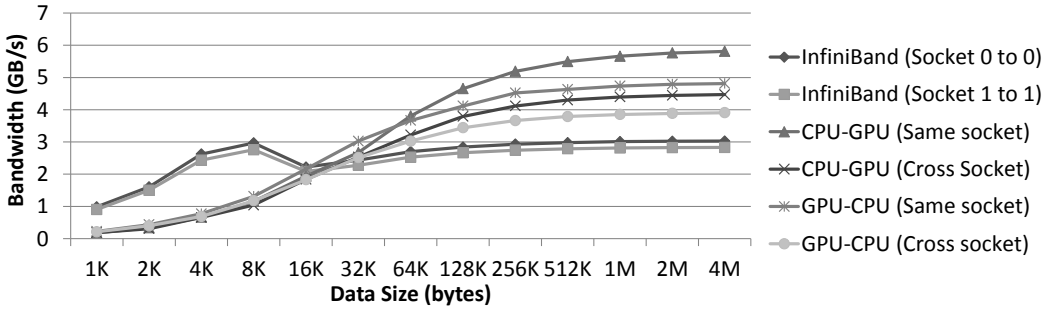


Figure 5.3: NUMA and PCIe affinity issues affecting the *effective* bandwidth of CPU-GPU and InfiniBand network transfers.

data transfer bandwidth can slow by as much as 49% (Figure 5.3). Similarly, the InfiniBand network bandwidth can slow by 7% because of the varying network controller affinity. Since the effective bandwidth can change dynamically and significantly, we also choose the pipeline parameters (packet size) dynamically at runtime for both the source and destination processes. We inspect a series of benchmark results; learn the dynamic system characteristics, such as the CPU socket binding; and then apply architecture-aware heuristics to choose the ideal transfer parameters for each communication request, all at runtime.

The sender sends a ready-to-send (RTS) message at the beginning of every MPI communication. The receiver sends a corresponding clear-to-send (CTS) message back, and then the sender begins to transfer the actual payload data. Thus, the sender encapsulates the local pipeline parameters within the RTS message and sends it across the network. The receiver inspects the sender's parameters and also the receiver's system characteristics (e.g., socket binding) and chooses the best pipeline parameters for the current communication transaction, based on our benchmark results and corresponding heuristics. The receiver then sends the CTS message back to the sender along with its chosen pipeline parameters. The sender uses the received pipeline parameters to perform data movement from the GPU, as before. In this approach, the two participating processes both first pick an initial pipeline configuration, then coordinate via RTS/CTS messages to converge on a single packet size depending on the effective bandwidth of the participating processes.

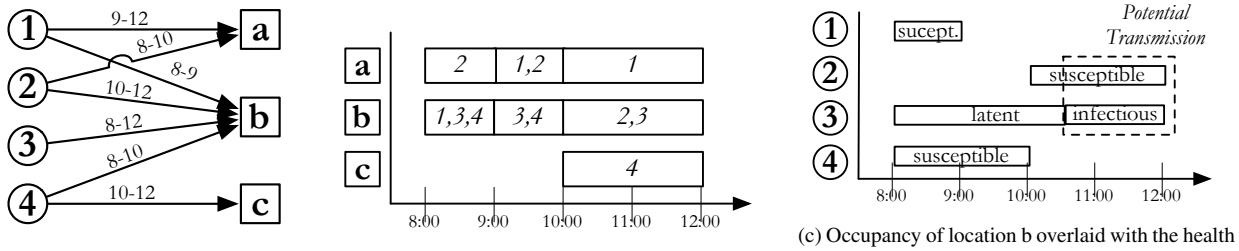
5.2.3 OpenCL Issues and Optimizations

In OpenCL, device data is encapsulated as a cl_mem object that is created by using a valid cl_context object. To transfer the data to/from the host, the programmer needs valid cl_device_id and cl_command_queue objects, which are all created by using the same context as the device data. At a minimum, the MPI interface for OpenCL communication requires the target OpenCL memory object, context, and device ID objects as parameters. The command queue parameter is optional and can be created by using the above parameters. Within the MPICH implementation, we either use the user-provided command queue or create several internal command queues for device-host data transfers. Within MPICH, we also create a temporary OpenCL buffer pool of pinned host-side memory for pipelining. However, OpenCL requires that the internal command queues and the pipeline buffers also be created by using the same context as the device data. Also, in theory, the OpenCL context could change for every MPI communication call, and so the internal OpenCL objects cannot be created at MPI_Init time. Instead, they must be created at the beginning of every MPI call and destroyed at the end of it.

The initialization of these temporary OpenCL objects is expensive and their repeated usage severely hurts performance. We cache the command queue and pipeline buffer objects after the first communication call and reuse them if the same OpenCL context and device ID are used for the subsequent calls, which is a very plausible scenario. If any future call involves a different context or device ID, we clear and replace our cache with the most recently used OpenCL objects. In this way, we can amortize the high OpenCL initialization cost across multiple calls and significantly improve performance. We use a caching window of one, which we consider to be sufficient in practice.

5.3 Application Case Studies

In this section, we first perform an in-depth analysis of the default MPI+GPU application design in scientific applications from computational epidemiology and seismology modeling. We identify the inherent data movement inefficiencies and show how MPI-ACC can be used to explore new design spaces and create novel application specific optimizations.



(a) Bipartite graph representation with people 1–4 and locations a–c.

(b) Temporal and spatial people-to-people contacts.

(c) Occupancy of location b overlaid with the health states of its occupants. The dotted area shows the time of possible disease transmission.

Figure 5.4: Computational epidemiology simulation model (figure adapted from [22]).

5.3.1 EpiSimdemics

GPU-EpiSimdemics [22, 24] is a high-performance, agent-based simulation program for studying the spread of epidemics through large-scale social contact networks and the co-evolution of disease, human behavior, and the social contact network. The participating entities in GPU-EpiSimdemics are *persons* and *locations*, which are represented as a bipartite graph (Figure 5.4a) and interact with each other iteratively over a predetermined number of iterations (or simulation days). The output of the simulation is the relevant disease statistics of the contagion diffusion, such as the total number of infected persons or an infection graph showing who infected whom and the time and location of the infection.

Phases

Each iteration of GPU-EpiSimdemics consists of two phases: *computeVisits* and *computeInteractions*. During the *computeVisits* phase, all the person objects of every processing element (or PE) first determine the schedules for the current day, namely, the locations to be visited and the duration of each visit. These *visit* messages are sent to the destination location's host PE (Figure 5.4a). Computation of the schedules is overlapped with the visit communication.

In the *computeInteractions* phase, each PE first groups the received visit messages by their target locations. Next, each PE computes the probability of infection transmission between every pair of spatially and temporally co-located people in its local location objects (Figure 5.4b), which determines the overall disease spread information of that location. The infection transmission function depends on the current health states (e.g., susceptible, infectious, latent)

of the people involved in the interaction (Figure 5.4c) and the transmissibility factor of the disease. These *infection* messages are sent back to the "home" PEs of the infected persons. Each PE, upon receiving its infection messages, updates the health states of the infected individuals, which will influence their schedules for the following simulation day. Thus, the messages that are computed as the output of one phase are transferred to the appropriate PEs as inputs of the next phase of the simulation. The system is synchronized by barriers after each simulation phase.

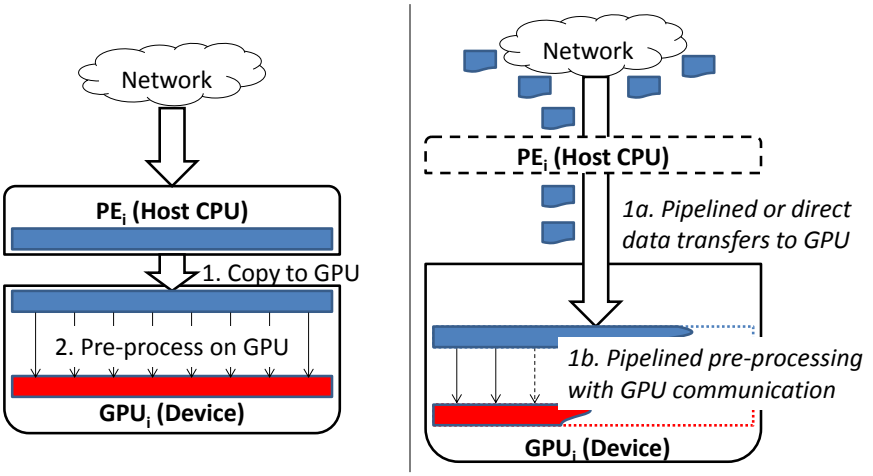
Computation-Communication Patterns and MPI-ACC-Driven Optimizations

In GPU-EpiSimdemics, each PE in the simulation is implemented as a separate MPI process. Also, the *computeInteractions* phase of GPU-EpiSimdemics is offloaded and accelerated on the GPU while the rest of the computations are executed on the CPU [24].[1] In accordance with the GPU-EpiSimdemics algorithm, the output data elements from the *computeVisits* phase (i.e., visit messages) are first received over the network, then merged, grouped, and preprocessed before the GPU can begin the *computeInteractions* phase of GPU-EpiSimdemics.

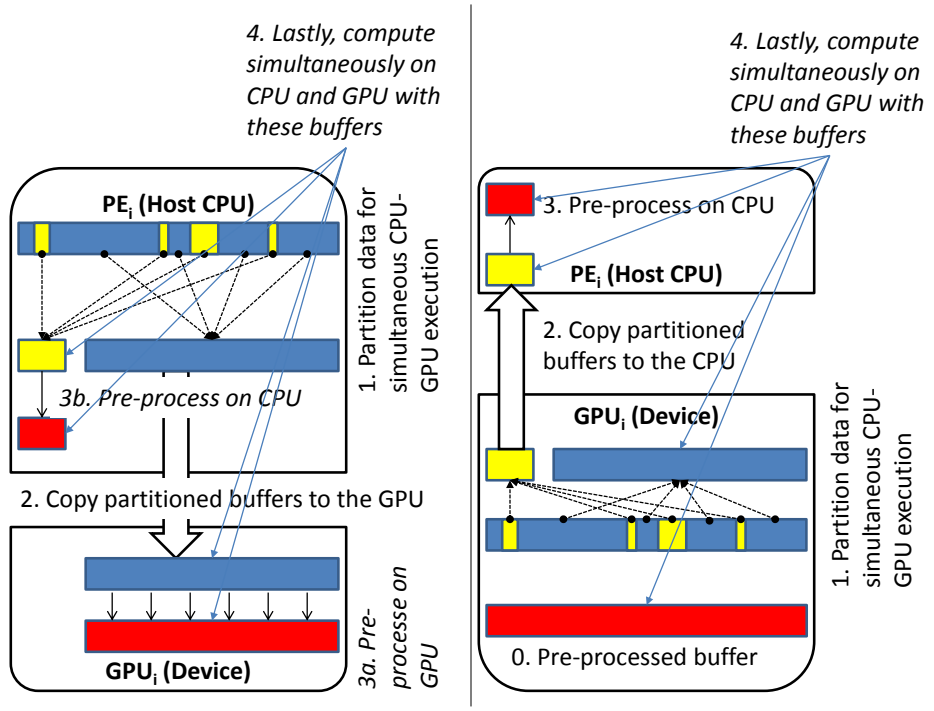
Moreover, there are two GPU computation modes depending on how the visit messages are processed on the GPUs: exclusive GPU computation, where all the visit messages are processed on the GPU, and cooperative CPU-GPU computation, where the visit messages are partitioned and concurrently processed on both the GPU and its host CPU. For each mode, we discuss the optimizations and tradeoffs. We also describe how MPI-ACC can be used to further optimize GPU-EpiSimdemics in both computation modes.

Exclusive CPU-GPU computation mode *Internode CPU-GPU data communication:* In the naïve data movement approach, each PE first receives all the visit messages in the CPU's main memory during the *computeVisits* phase, then transfers the aggregate data to the local GPU (device) memory across the PCIe interface at the beginning of the *computeInteractions* phase. The typical all-to-all or scatter/gather type of operation is not feasible because the number of pairwise visit message exchanges is not known beforehand in GPU-EpiSimdemics. Thus, each PE preallocates and registers fixed-sized *persistent* buffer fragments with the `MPI_Recv_init` call and posts the receive requests by

[1] The current implementation of GPU-EpiSimdemics assumes one-to-one mapping of GPUs to MPI processes.



(a) Exclusive GPU computation mode. Left: Manual MPI+CUDA optimizations, where the *visit* messages are received on the host, then copied to the device for preprocessing. Right: New MPI-ACC–enabled optimizations, where the *visit* messages are transparently pipelined into the device and preprocessing is overlapped.



(b) Cooperative CPU-GPU computation mode. Left: Manual MPI+CUDA optimizations, where data partitioning happens on the CPU. Right: New MPI-ACC–enabled optimizations, where the data distribution happens on the GPU. The preprocessing of the GPU data is still overlapped with communication.

Figure 5.5: Creating new optimizations for GPU-EpiSimdemics using MPI-ACC.

subsequently calling `MPI_Start_all`. Whenever a buffer fragment is received, it is copied into a contiguous visit vector in the CPU's main memory. The *computeInteractions* phase of the simulation first copies the aggregated visit vector to the GPU memory. While the CPU-CPU communication of visit messages is somewhat overlapped with their computation on the source CPUs, the GPU and the PCIe interface will remain idle until the visit messages are completely received, merged, and ready to be transferred to the GPU.

Preprocessing phase on the GPU: As a preprocessing step in the *computeInteractions* phase, we modify the data layout of the visit messages to be more amenable to the massive parallel architecture of the GPU [24]. Specifically, we unpack the visit message structures to a 2D time-bin matrix, where each row of the matrix represents a person-location pair and the cells in the row represents fixed time slots of the day: that is, each visit message corresponds to a single row in the person-timeline matrix. Depending on the start time and duration of a person's visit to a location, the corresponding row cells are marked as *visited*. The preprocessing logic of data unpacking is implemented as a separate GPU kernel at the beginning of the *computeInteractions* phase. The matrix data representation enables a much better SIMDization of the *computeInteractions* code execution, which significantly improves the GPU performance. However, we achieve the benefits at the cost of a larger memory footprint for the person-timeline matrix, as well as a computational overhead for the data unpacking. *MPI-ACC–enabled optimizations:* In the basic version of GPU-EpiSimdemics, the GPU remains idle during the internode data communication phase of *computeVisits*, whereas the CPU remains idle during the preprocessing of the *computeInteractions* phase on the GPU. With MPI-ACC, during the *computeVisits* phase, we transfer the visit message fragments from the source PE directly to the destination GPU's device memory. Internally, MPI-ACC may pipeline the internode CPU-GPU data transfers via the host CPU's memory or use direct GPU transfer techniques (e.g., GPUDirect RDMA), if possible, but these details are hidden from the programmer. The fixed-sized *persistent* buffer fragments are now preallocated *on the GPU* and registered with the `MPI_Recv_init` call, and the contiguous visit vector is not created in the GPU memory itself. Furthermore, as soon as a PE receives the visit buffer fragments on the GPU, we immediately launch small GPU kernels that preprocess on the received visit data, that is, unpack the partial visit messages to the 2D data matrix layout (Figure 5.5). These preprocessing kernels execute asynchronously with respect to the CPU in a pipelined fashion and thus are completely overlapped by the visit

data generation on the CPU and the internode CPU-GPU data transfers. In this way, the data layout transformation overhead is completely hidden and removed from the *computeInteractions* phase. Moreover, the CPU, GPU, and the interconnection networks are all kept busy, performing either data transfers or the preprocessing execution.

MPI-ACC's internal pipelined CPU-GPU data transfer largely hides the PCIe transfer latency during the *computeVisits* phase. However, it still adds a non-negligible cost to the overall communication time when compared with the CPU-CPU data transfers of the default MPI+GPU implementation. However, our experimental results show that the gains achieved in the *computeInteractions* phase due to the preprocessing overlap outweigh the communication overheads of the *computeVisits* phase for all system configurations and input data sizes.

Advanced MPI+GPU optimizations without using MPI-ACC: The above discussed optimizations can also be implemented at the application level without using MPI-ACC, as follows. The fixed-sized *persistent* receive buffer fragments are pre-allocated on the CPU itself and registered with the `MPI_Recv_init` call, but the contiguous visit vector resides in GPU memory. Whenever a PE receives a visit buffer fragment on the CPU, we immediately enqueue an asynchronous CPU-GPU data transfer to the contiguous visit vector and also enqueue the associated GPU preprocessing kernels.

However, to enable asynchronous CPU-GPU data transfers, the receive buffers have to be nonpageable (pinned) memory. Moreover, since the number of visit message exchanges is not known beforehand to the application, each PE creates a receive buffer fragment corresponding to every other participating PE in the simulation, i.e. the pinned memory footprint increases with the number of processes. This design reduces the available pageable CPU memory which could lead to poor CPU performance [9]. The pinned memory management logic can be implemented at the application level in a couple of ways, as follows. In the first approach, the pinned memory pool is created before the *computeVisits* phase begins and is destroyed once the phase finishes, but the memory management routines are invoked every simulation iteration. While this approach is relatively simple to implement, repeated memory management leads to significant performance loss. In the second approach, the pinned memory pool is created once before the main simulation loop and destroyed after the loop ends, which avoids the performance overhead of repeated memory management. However, this approach leads to a more complex application design because the programmer has

to explicitly and efficiently reuse the pinned memory. Moreover, the large and growing pinned memory allocation reduces the available pageable CPU memory not only for the *computeVisits* phase, but also for the other phases of the simulation, including *computeInteractions*. We discuss the performance tradeoffs of the above two manual memory management techniques in section 5.4.2.

On the other hand, MPI-ACC helps the programmer by enabling automatic and efficient memory management techniques. While the MPI-ACC–based solution also manages persistent receive buffers, it does so on the GPU's memory, which is an order of magnitude faster than managing the host-side pinned memory. MPI-ACC internally creates and manages a *constant* pool of pinned memory during `MPI_Init` and automatically reuses it for all CPU-GPU transfers, thereby enabling better scaling and easier programmability. Moreover, MPI-ACC exposes a natural interface to communicate with the target device (CPU or GPU), without treating CPUs as explicit communication relays.

Cooperative CPU-GPU computation mode The exclusive GPU computation mode achieved significant overlap of communication with computation during the preprocessing phase. When the infection calculation of the *computeInteractions* phase was executed on the GPU, however, the CPU remained idle. On the other hand, in the cooperative computation mode, all the incoming visit messages are partitioned and processed concurrently on the GPU and its host CPU during the *computeInteractions* phase, an approach that gives better parallel efficiency. Again, we present three optimizations with their tradeoffs.

Basic MPI+GPU with data partitioning on CPU: In the MPI+GPU programming model, the incoming visit vector on the CPU is not transferred in its entirety to the GPU. Instead, the visit messages are first grouped by their target locations into buckets. Within each visit group, the amount of computation increases quadratically with the group size because it is an all-to-all person-person interaction computation within a location. Each visit group can be processed independently of the others but has to be processed by the same process or thread (CPU) or thread block (GPU). Therefore, data partitioning in GPU-EpiSimdemics is done at the granularity of *visit groups* and not individual visit messages.

At a high level, the threshold for data partitioning is chosen based on the computational capabilities of the target

processors (e.g., GPUs get more populous visit groups for higher concurrency) so that the execution times on the CPU and the GPU remain approximately the same. The visit messages that are marked for GPU execution are then grouped and copied to the GPU device memory, while the CPU visit messages are grouped and remain on the host memory (Figure 5.5b).

Preprocessing and computation phases: In this computation mode, preprocessing, in other words, unpacking the visit structure layout to the person-timeline matrix layout, is concurrently executed on the CPU and GPU on their local visit messages (Figure 5.5b). Next, the CPU and GPU simultaneously execute *computeInteractions* and calculate the infections.

MPI-ACC–enabled optimizations with data partitioning on GPU: In the MPI-ACC model, the computation of the *computeInteractions* phase is executed on the CPU and GPU concurrently. While this approach leads to better resource utilization, the data partitioning logic itself and the CPU-GPU data transfer of the partitioned data add nontrivial overheads that may offset the benefits of concurrent execution. However, our results in Section 5.4.2 indicate that executing the data partitioning logic *on the GPU* is about 53% faster than on the CPU because of the GPU's higher memory bandwidth. With MPI-ACC, the visit vector is directly received or pipelined into the GPU memory, and the data partitioning logic is executed on the GPU itself. Next, the CPU-specific partitioned visit groups are copied *to the CPU* (Figure 5.5b). As a general rule, if the GPU-driven data partitioning combined with the GPU-to-CPU data transfer performs better than the CPU-driven data partitioning combined with CPU-to-GPU data transfer, then GPU-driven data partitioning is a better option. Our experimental results (Section 5.4.2) indicate that for GPU-EpiSimdemics, the MPI-ACC enabled GPU-driven data partitioning performs better than the other data partitioning schemes.

The preprocessing phase *on the GPU* is still overlapped with the internode CPU-GPU communication by launching asynchronous GPU kernels, just like the exclusive GPU mode, thereby largely mitigating the preprocessing overhead. While this approach could lead to redundant computations for the CPU-specific visit groups on the GPU, the corresponding person-timeline matrix rows can be easily ignored in the subsequent execution phases. This approach will create some unnecessary memory footprint on the GPU; however, the benefits of overlapped preprocessing outweigh the issue of memory overuse. On the other hand, the preprocessing *on the CPU* is executed only after the data parti-

tioning and GPU-to-CPU data transfer of CPU-specific visit groups. This step appears on the critical path and cannot be overlapped with any other step, but it causes negligible overhead for GPU-EpiSimdemics because of the smaller data sets for the CPU execution.

Advanced MPI+GPU with data partitioning on GPU: GPU-driven data partitioning can also be implemented without using MPI-ACC, where the visits vector is created on the GPU and the preprocessing stage is overlapped by the local CPU-GPU data communication, similar to the advanced MPI+GPU optimization of the exclusive GPU computation mode. The data partitioning on the GPU and the remaining computations follow from the MPI-ACC–enabled optimizations. As in the GPU-exclusive computation mode, however, the pinned memory footprint increases with the number of processes, which leads to poor CPU performance and scaling. Moreover, from our experience, the back-and-forth CPU-GPU data movement in the GPU-driven data partitioning optimization seems convoluted without a GPU-integrated MPI interface. On the other hand, MPI-ACC provides a natural interface for GPU communication, which encourages application developers to explore new optimization techniques such as GPU-driven data partitioning and to evaluate them against the default and more traditional CPU-driven data partitioning schemes.

5.3.2 FDM-Seismology

FDM-Seismology is our MPI+GPU implementation of an application that models the propagation of seismological waves using the finite-difference method (FDM) by taking the earth's velocity structures and seismic source models as input [53]. The application implements a parallel velocity-stress, staggered-grid finite-difference method 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 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. This strategy helps simulate unbounded domains. In our application, PML (perfectly matched layers) absorbers [23] are being 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 [53].

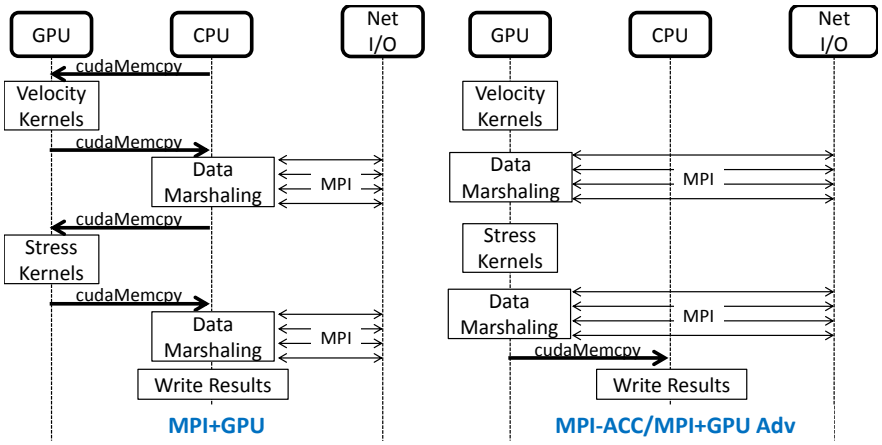


Figure 5.6: Communication-computation pattern in the FDM-Seismology application. Left: basic MPI+GPU execution mode with data marshaling on CPU. Right: execution modes with data marshaling on GPU. MPI-ACC automatically communicates the GPU data; MPI+GPU Adv case explicitly stages the communication via CPU.

Computation-Communication Patterns

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 CPUs (or nodes). This domain decomposition technique helps 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 5.6). Each processor updates its own wavefields after receiving all its neighbors' wavefields.

These computations are run for multiple 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, and then unpacked at the receiver to update its local data.

GPU Acceleration of FDM-Seismology

Here, we describe a couple of GPU execution modes of FDM-Seismology.

MPI+GPU with data marshaling on CPU (MPI+GPU): Our GPU-accelerated version of FDM-Seismology performs the velocity and stress computations as GPU kernels. 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 interface and then transfers the individual wavefields over MPI to the neighboring nodes (Figure 5.6). All the data-marshaling operations and small post-communication 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): In this execution mode, the data-marshaling operations are moved to the GPU to leverage the faster GDDR5 memory module and the massively parallel GPU architecture. As a consequence, the CPU-GPU bulk data transfers before and after each velocity-stress computation kernel are completely 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 (Figure 5.6).

MPI-ACC–Enabled Optimizations

GPU-based data marshaling suffers from the following disadvantage in the absence of GPU-integrated MPI. All data-marshaling steps are separated by MPI communication, and each data-marshaling step depends on the previously marshaled data *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., GPUDirect RDMA), we will have to largely rewrite the FDM-Seismology communication code to achieve the expected performance.

With MPI-ACC as the communication library, we still perform data marshaling on the GPU, but communicate the marshaled data directly to and from the GPU without explicitly using the CPU for data staging. Also, the bulk transfer of data still happens only once at the end of each iteration to write the results to a file. However, the data-marshaling step happens multiple times during a single iteration and consequently, the application launches a series of GPU kernels. While consecutive kernels entail launch and synchronization overhead per kernel invocation, the benefits of faster data marshaling on the GPU and optimized MPI communication outweigh the kernel overheads.

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 benefit from the performance upgrades that are made available by the subsequent GPU-integrated MPI implementations (e.g., support for GPUDirect RDMA).

5.4 Evaluation

In this section, we describe our experimental setup followed by the performance evaluation of MPI-ACC via latency microbenchmarks. Next, we demonstrate the efficacy of the MPI-ACC–enabled optimizations in GPU-EpiSimdemics and FDM-Seismology. Finally, using both microbenchmarks and GPU-EpiSimdemics, we study the impact of shared resource (hardware and software) contention on MPI-ACC's communication performance.

We conducted our experiments on *HokieSpeed*, a state-of-the-art, 212-teraflop hybrid CPU-GPU supercomputer housed

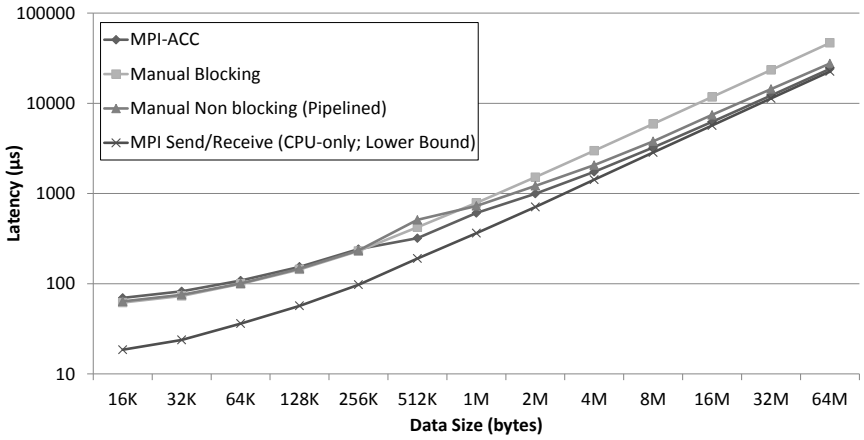


Figure 5.7: Internode communication latency for GPU-to-GPU (CUDA) data transfers. Similar performance is observed for OpenCL data transfers. The chosen pipeline packet size for MPI-ACC is 256 KB.

at Virginia Tech. Each HokieSpeed node contains two hex-core Intel Xeon E5645 CPUs running at 2.40 GHz and two NVIDIA Tesla M2050 GPUs. The host memory capacity is 24 GB, and each GPU has a 3 GB device memory. The internode interconnect is QDR InfiniBand. We used up to 128 HokieSpeed nodes and both GPUs per node for our experiments. We used the GCC v4.4.7 compiler and CUDA v5.0 with driver version 310.19.

5.4.1 Microbenchmark Analysis

Impact of Pipelined Data Transfer

In Figure 5.7 we compare the performance of MPI-ACC with the manual blocking and manual pipelined implementations. Our internode GPU-to-GPU latency tests show that MPI-ACC is better than the manual blocking approach by up to 48.3% and is up to 18.2% better than the manual pipelined implementation, especially for larger data transfers. The manual pipelined implementation performs poorly because of the repeated handshake messages that are sent back and forth across the network before the data transfer. For message sizes that are smaller than the pipeline packet size, we show that the performance of MPI-ACC is comparable to the manual approaches. This is because we use the MPI-ACC pipelining logic only to transfer data that is larger than one pipeline packet size, and we fall back to the default blocking approach for smaller data sizes.

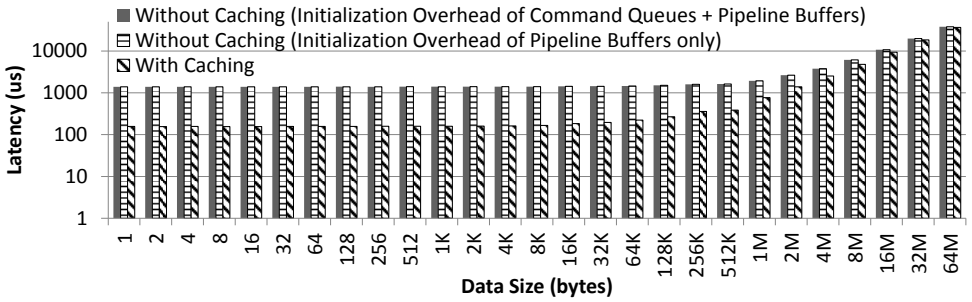


Figure 5.8: MPI-ACC performance with and without OpenCL object caching.

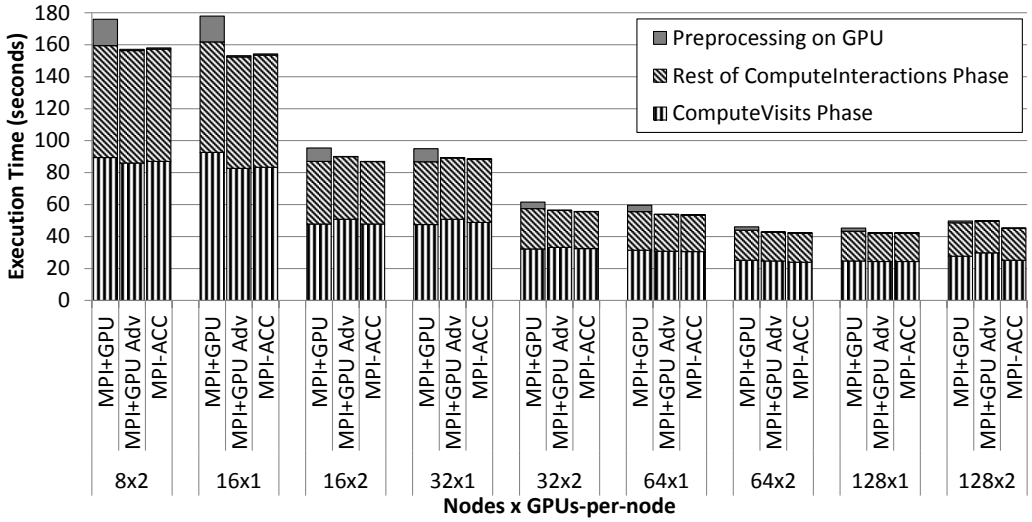
Impact of OpenCL Object Caching

Figure 5.8 shows that the OpenCL caching optimization improves the internode GPU-to-GPU communication latency from 3% for larger data sizes (64 MB) to 88.7% for smaller data sizes ($<$ 256 KB). Even where the programmers provide their custom command queue, the pipeline buffers still have to be created for every MPI communication call, and so caching improves performance.

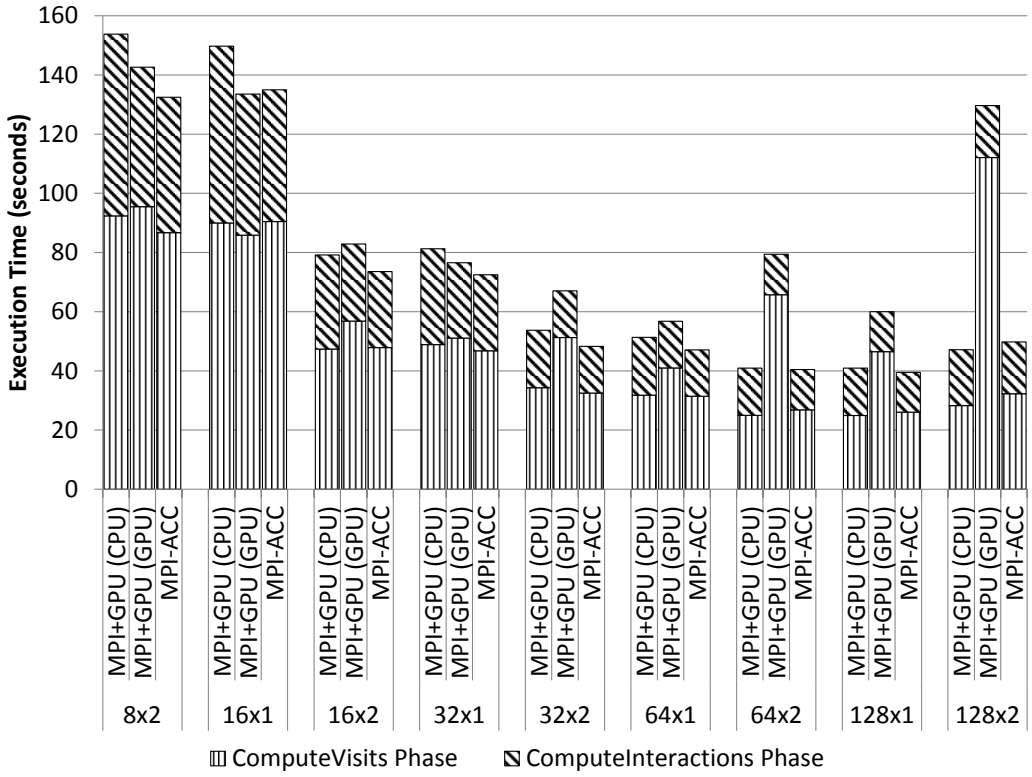
5.4.2 Case Study Analysis: EpiSimdemics

We compare the combined performance of all the phases of GPU-EpiSimdemics (*computeVisits* and *computeInteractions*), with and without the MPI-ACC–driven optimizations discussed in Section 5.3.1. We choose different-sized input data sets from synthetic populations from two U.S. states: Washington (WA) with a population of 5.7 million and California (CA) with a population of 33.1 million. We also vary the number of compute nodes from 8 to 128 and the number of GPU devices between 1 and 2. We begin from the smallest node-GPU configuration that can fit the entire problem in the available GPU memory.

Our results in Figure 5.9 indicate that in the exclusive GPU-computation mode, our MPI-ACC–driven optimizations perform better than the basic blocking MPI+GPU implementations by an average of 9.2% and by up to 13.3% for WA. The MPI-ACC–driven solution and the advanced manual pipelined MPI+GPU implementations have performances within 1.7% of each other. Note that the advanced MPI+GPU implementation uses the manual pinned memory management techniques that we implemented at the application level, which achieves better performance but with much



(a) Exclusive GPU mode for Washington.



(b) Cooperative CPU-GPU mode for Washington.

Figure 5.9: Execution profile of GPU-EpiSimdemics over various node configurations. The x-axis increases with the total number of MPI processes P, where P = Nodes * GPUs.

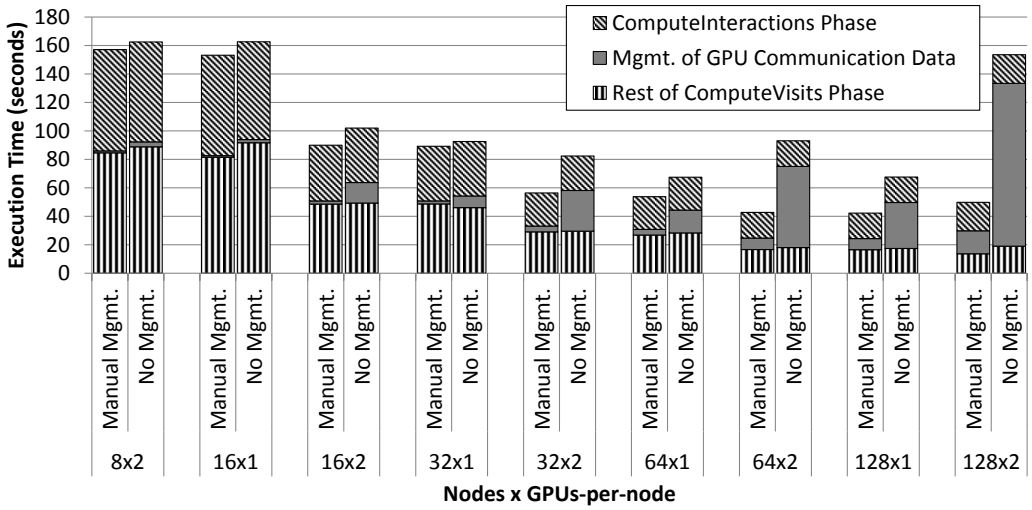


Figure 5.10: Analysis of the data management complexity vs. performance trade-offs. Manual data management achieves better performance at the cost of high code complexity. No explicit data management has simpler code but performs poorly.

more complex code.

For both the MPI-ACC and advanced MPI+GPU implementations, the preprocessing step (data unpacking) of the *computeInteractions* phase is completely overlapped with the CPU to remote GPU communication for all node configurations. For larger node configurations, however, the local operating data set in the *computeInteractions* phase becomes smaller, which means that the basic MPI+GPU solution takes less time to execute the preprocessing stage. So, the absolute gains over the basic MPI+GPU solution, which is achieved by hiding the preprocessing step, get diminished while the relative gains still hold. Note that the data transfer optimizations within MPI-ACC or any other GPU-integrated MPI, by themselves, do not impact the performance gains. In contrast, MPI-ACC enables the developer to create newer optimizations for better resource utilization.

Data management complexity vs. performance tradeoffs

While the advanced MPI+GPU implementation achieved comparable performance to the MPI-ACC–based solution, it put the burden of explicit data management on the application programmer. We discussed in section 5.3.1 that on the other hand, the user can write simpler code and avoid explicit data management, but has to repeatedly create and

Table 5.1: Analyzing the memory allocation costs. Note: each CUDA context is managed by a separate process.

Number of CUDA Contexts	`malloc`	`cudaMallocHost`
1 (normal allocation)	1.10μs	1.39μs
2 (parallel allocation)	1.11μs	2.67μs

destroy the receive buffers for every simulation iteration and lose performance. Figure 5.10 shows the performance tradeoffs of the above two approaches. We observe that explicit data management is better for all node configurations and can achieve up to $4.5\times$ performance improvement. Without data management, the pinned memory footprint of the receive buffers increases with the number of MPI processes, which entails bigger performance losses for larger nodes. To quantify the degree of performance loss, we measured the individual memory allocation costs via simple microbenchmarks and found that CUDA's pinned memory allocator (`cudaMallocHost`) is about 26% slower than the vanilla CPU memory allocator (`malloc`) for single CUDA contexts (table 5.1). We also see that the pinned memory allocation cost increases linearly with the number of GPUs or CUDA contexts, whereas memory management in multiple processes and CUDA contexts should ideally be handled independently in parallel. Consequently, in figure 5.10, we see that for the same number of MPI processes, the node configuration with two MPI processes (or GPUs) per node performs worse than the node with a single MPI process, e.g. 64×2 configuration is slower than the the 128×1 one. Thus, efficient pinned memory management is essential for superior performance but it comes at the cost of significant programmer effort.

Discussion: The basic MPI+GPU solution has the preprocessing overhead but does not have significant memory management issues. While the advanced MPI+GPU implementation gains from hiding the preprocessing overhead, it loses from either nonscalable pinned memory management or poor programmer productivity. On the other hand, MPI-ACC provides a more scalable solution by (1) automatically managing a fixed-size pinned buffer pool for pipelining and (2) creating them just once at `MPI_Init` and destroying them at `MPI_Finalize`. MPI-ACC thus gains from both hiding the preprocessing overhead and efficient pinned memory management. MPI-ACC decouples the lower-level memory management logic from the high-level simulation implementation, thereby enabling both performance and productivity.

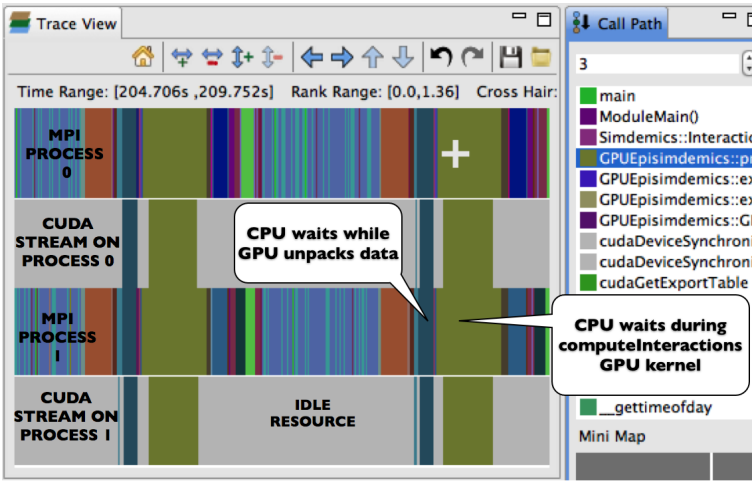
Analysis of resource utilization using HPCToolkit

HPCTOOLKIT [10] is a sampling based performance analysis toolkit capable of quantifying scalability bottlenecks in parallel programs. In this paper, we use an extension of HPCTOOLKIT that works on hybrid (CPU-GPU) codes; the extension uses a combination of sampling and instrumentation of CUDA code to accurately identify regions of low CPU/GPU utilization. HPCTOOLKIT presents program execution information through two interfaces: `hpcviewer` and `hpctraceviewer`. `Hpcviewer` associates performance metrics with source code regions including lines, loops, procedures, and calling contexts. `Hpctraceviewer` renders hierarchical, timeline-based visualizations of parallel program executions.

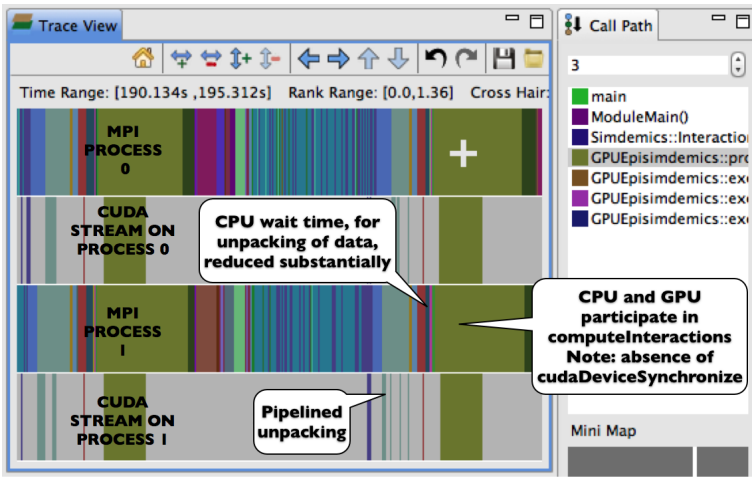
In Figure 5.11, we present snapshots of the execution profile of GPU-EpiSimdemics from the `hpctraceviewer` tool of HPCTOOLKIT. Figure 5.11a depicts the application without the MPI-ACC-driven optimizations. The timeline information of all CPU processes and their corresponding CUDA streams is presented by the `hpctraceviewer` tool. The *call path* pane on the right represents the call stack of the process/stream at the current crosshair position. Although we study a 32-process execution, we zoom in and show only the 0th and 1st processes and their associated CUDA streams, because the other processes exhibit identical behavior.

The figure depicts two iterations of the application, where a couple of *computeInteractions* phases, with the corresponding GPU activity, are surrounding a *computeVisits* phase, where there is no GPU activity. The GPU idle time during the *computeVisits* phase can be reduced by offloading parts of the *computeVisits* computation to the GPU; but that is beyond the scope of this paper.

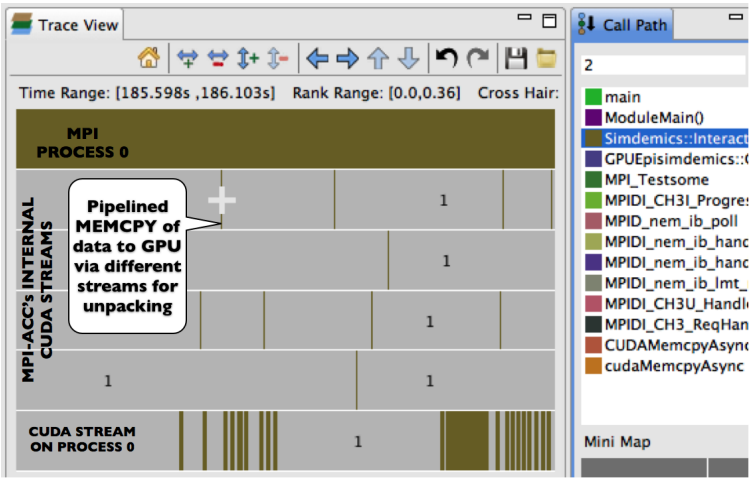
In the basic hybrid MPI+GPU programming model, the application launches kernels on the *default* CUDA stream for the *computeInteractions* phase, including the preprocessing (or data unpacking) and the main infection processing stages. In the figure, we can see a corresponding set of bars on the default CUDA stream in the *computeInteractions* phase, which denote the following: (1) a small, negligible sliver showing `cudaMemcpy` of the visit messages from the CPU to the GPU; (2) a medium-sized bar showing preprocessing (or data unpacking) on the GPU; and (3) a thick band showing the main infection computation kernel. The figure thus helps identify two distinct issues and opportunities



(a) Manual MPI+CUDA optimizations. The *visit* messages are first received on the CPU and copied to the device; then the preprocessing (unpacking) takes place on the GPU.



(b) MPI-ACC optimizations. The *visit* messages are received directly in the device. Preprocessing (unpacking) on the GPU is pipelined and overlapped with data movement to the GPU. This leads to negligible CPU waiting while the GPU preprocesses/unpacks the data.



(c) MPI-ACC optimizations. This figure combines (b) with activity occurring on other streams. MPI-ACC employs multiple streams to push the data to the device asynchronously, while the application initiates the unpacking of data.

Figure 5.11: Analysis of MPI-ACC–driven optimizations using HPCTOOKIT. Application case study: GPU-EpiSimdemics.

for performance improvement in the *computeInteractions* phase of GPU-EpiSimdemics.

1. The thick band on the CUDA stream representing the main kernel of the *computeInteractions* phase has a corresponding thick `cudaDeviceSynchronize` band on the CPU side; that is, the CPU is idle while waiting for the GPU, thus indicating that some work from the GPU can be offloaded to the CPU.

2. The medium-sized bar on the CUDA stream representing the preprocessing (data unpacking) step has a corresponding `cudaDeviceSynchronize` bar on the CPU, which indicates that the CPU can start offloading the data to be unpacked to the GPU in stages, thus overlapping data transfers to the GPU with their unpacking on the GPU.

We resolve the first issue by using the cooperative CPU-GPU computation mode. The second issue is resolved in both the cooperative and the exclusive GPU modes. We use MPI-ACC to pipeline the data unpacking before the *computeInteractions* phase by overlapping it with the *computeVisits* phase. We use a custom CUDA stream to execute the preprocessing kernel so that we can achieve an efficient overlap between the host-to-device (H2D) data transfers within MPI-ACC and the preprocessing kernel of GPU-EpiSimdemics. Figure 5.11b, which represents HPCTOOLKIT's trace view on applying these optimizations, shows that the time wasted by the CPU in `cudaDeviceSynchronize` while the GPU unpacked the data has disappeared (compared with Figure 5.11a). This reduction in the CPU idle time characterizes the success of the MPI-ACC–driven optimizations.

Figure 5.11c shows a zoomed-in version of Figure 5.11b, where we can see the internal helper streams that are created within MPI-ACC along with the custom CUDA stream of one of the processes (only a subset of MPI-ACC's internal streams is shown here for brevity). While the GPU kernels of the *computeInteractions* phase are executed on the application's custom stream, the staggered bars in the MPI-ACC's internal streams represent the pipelined data transfers before the unpacking stage, thus showing efficient use of concurrency via multiple GPU streams.

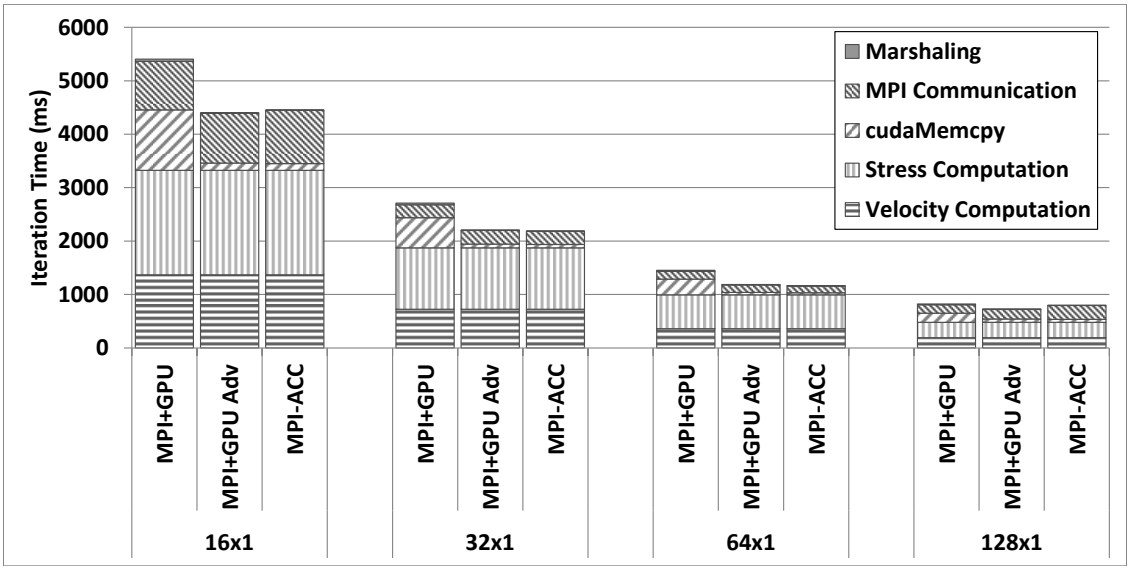


Figure 5.12: Analyzing the FDM-Seismology application with the larger input data (Dataset-2). Note: MPI Communication refers to CPU-CPU data transfers for the MPI+GPU and MPI+GPU Adv cases and GPU-GPU (pipelined) data transfers for the MPI-ACC case.

5.4.3 Case Study Analysis: FDM-Seismology

In this section, we analyze the performance of the different phases of the FDM-Seismology application and evaluate the effect of MPI-ACC optimizations on the application. We vary the nodes from 2 to 128 with 1 GPU per node and use small and large datasets as input. Our scalability experiments begin from the smallest number of nodes required to fit the given data in the GPU memory. For the larger input data (i.e., Dataset-2), the size of the MPI transfers increases by $2\times$ while the size of data to be marshaled increases by $4\times$ when compared with the smaller Dataset-1.

Figure 5.12 shows the performance of the FDM-Seismology application, with and without the MPI-ACC–enabled designs. We report the average wall clock time across all the processes because the computation-communication costs vary depending on the virtual location of the process in the application's structured grid representation. The application's running time is mainly composed of velocity and stress computations ($>60\%$), which does not change for all the three application designs.

In the basic MPI+GPU case, we perform both data-marshaling operations and MPI communication from the CPU. So, the application has to move large wavefield data between the CPU and the GPU for data marshaling and MPI

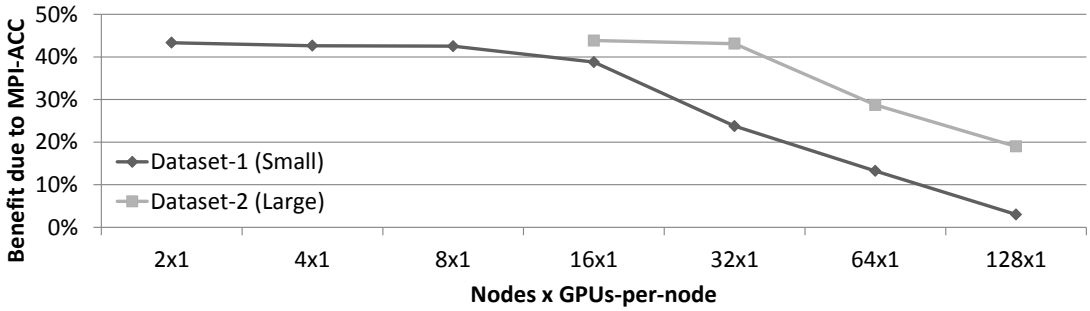


Figure 5.13: Scalability analysis of FDM-Seismology application with two datasets of different sizes. The baseline for speedup is the naïve MPI+GPU programming model with CPU data marshaling.

communication after every stress and velocity computation phase over every iteration. In the MPI+GPU Adv and MPI-ACC–driven scenarios, we perform data marshaling on the GPU itself and so, smaller sized wavefield data is transferred from GPU to CPU only once per iteration for output generation. By performing data marshaling on the GPU, we avoid the large bulk CPU-GPU data transfers and improve the overall performance over the basic MPI+GPU design with data marshaling on the CPU.

Scalability analysis: Figure 5.13 shows the performance improvement due to the MPI-ACC–enabled GPU data marshaling strategy over the basic MPI+GPU implementation with CPU data marshaling. We see that the performance benefits due to the GPU data marshaling decrease with increasing number of nodes, for the reasons noted below. For a given dataset, the per-node data size decreases with increasing number of nodes. This reduces the costly CPU-GPU bulk data transfers (Figure 5.12) and thus minimizes the overall benefits of GPU-based data marshaling itself. Also, for larger number of nodes, 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 MPI communication of the MPI+GPU and MPI+GPU Adv implementations will have less overhead than the pipelined GPU-to-GPU MPI communication of the MPI-ACC–enabled design. 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.

5.4.4 Analysis of Contention

In this section, we provide some insights into the scalable design of MPI-ACC. Specifically, we show that MPI-ACC is designed to work concurrently with other existing GPU workloads with minimum contention; that is, one should be able to perform MPI-ACC GPU-GPU communication and other user-specified GPU tasks (kernel or data transfers) simultaneously with minimum performance degradation for both tasks. We use microbenchmarks as well as the GPU-EpiSimdemics application for our evaluation study.

Sources of contention: NVIDIA Fermi GPUs have one hardware queue each for enqueueing GPU kernels, device-to-host (D2H) data transfers, and host-to-device (H2D) data transfers. Operations on different hardware queues can potentially overlap. For example, a GPU kernel can overlap with H2D and D2H transfers simultaneously. However, operations enqueued to the same hardware queue will be processed serially. If a GPU task oversubscribes a hardware queue by aggressively enqueueing multiple operations of the same type, then it can severely slow down other GPU tasks contending to use the same hardware queue.

GPU streams are software workflow abstractions for a sequence of operations that execute in issue-order on the GPU. Stream operations are directed to the appropriate hardware queue depending on the operation type. Operations from different streams can execute concurrently and may be interleaved, whereas operations within the same stream are processed serially, leading to software contention.

In summary, contention among GPU operations can be of two types: hardware contention, where one or more hardware queues of the GPU are oversubscribed by the same type of operation, or software contention, where different types of operations may be issued but to the same GPU stream. In MPI-ACC, we have carefully minimized both types of contention by staggered enqueueing of H2D and D2H operations to different GPU streams, thereby enabling maximum concurrency.

Microbenchmark design: We extended the SHOC [31] benchmark suite's `contention-mt` application for the microbenchmark study. The benchmark creates two MPI processes, each on a separate node and controlling the two local GPUs. Each MPI process is also dual-threaded and concurrently runs one task per thread, where task-0 by

thread-0 does MPI-ACC point-to-point GPU-to-GPU communication with the other process and task-1 by thread-1 executes local non-MPI GPU tasks, such as compute kernels or H2D and D2H data transfer loops. CUDA allows the same GPU context to be shared among all the threads (tasks) in the process. We share the local GPU between both tasks. To measure the contention impact, we first execute tasks 0 and 1 independently without contention and then execute them concurrently to induce contention. Each task is run for 100 loop iterations. We measure and report the performance difference between the tasks' independent and concurrent runs as the incurred contention.

Quantifying the Hardware Contention

MPI-ACC uses the D2H and H2D hardware queues of the GPU for send and receive, respectively. In theory, MPI-ACC communication can overlap with kernel invocations or other data transfer operations in the opposite direction, that is, using the other data transfer queue. However, MPI-ACC can cause contention with another data transfer operation in the same direction. For example, `MPI_Send` can contend with a concurrent D2H data transfer. MPI-ACC operations can also potentially contend with the on-device memory controller. For example, `MPI_Send` or `MPI_Recv` can slow a global-memory-intensive kernel that is accessing the same memory module. In this section, we quantify and evaluate the global memory and PCIe contention effects.

Global memory contention analysis: We study the impact of global memory contention by executing MPI-ACC operations in task-0 and global memory read/write benchmarks in task-1 with custom CUDA streams. Our experiments indicate that the performance drop due to contention in the MPI-ACC communication is about 4%, whereas the global memory kernels slow by about 8%. The average MPI-ACC call runs longer than an average global memory access, so MPI-ACC has less relative performance reduction. On the other hand, the performance impact of MPI-ACC on on-chip (local) memory accesses and simple computational kernels using custom CUDA streams is less, where the performance degradation in the MPI-ACC communication is about 3% and the GPU workloads do not have any noticeable slowdown. Figure 5.14 depicts the performance slowdown of the MPI-ACC communication due to contention.

PCIe contention analysis with data transfers in the opposite direction: We study the impact of PCIe contention by

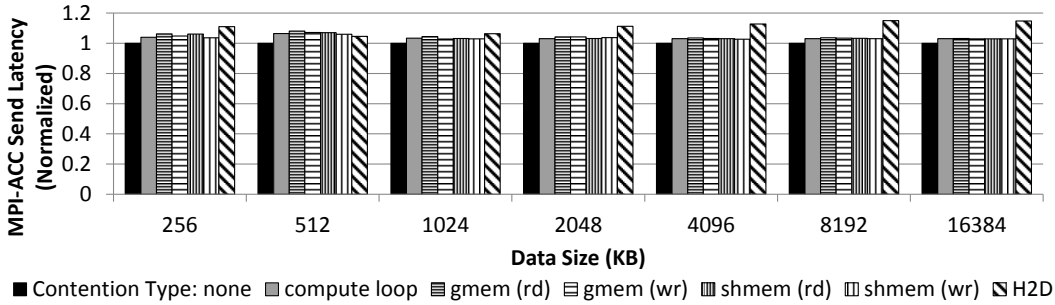


Figure 5.14: Contention impact of concurrent `MPI_Send` and local GPU operations (compute kernels, global memory read/write, shared memory read/write and host-to-device (H2D) data transfers).

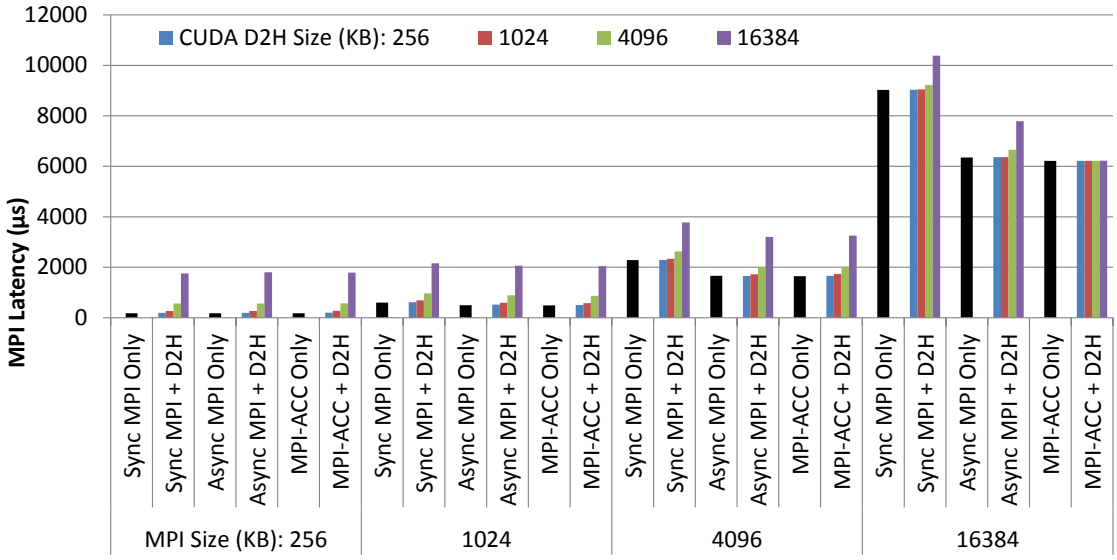
having task-0 perform `MPI_Send` or `MPI_Recv` communication operations with GPU-0, while task-1 executes H2D or D2H calls. This approach gives four different task combinations, of which two combinations perform bidirectional data transfers and two combinations transfer data in the same direction. Here, we report the results by running `MPI_Send` (task-0) concurrently with H2D and D2H transfers on the same GPU (task-1). The contention analysis of `MPI_Recv` with H2D and D2H transfers is identical. If task-0 and task-1 perform bidirectional data transfers and use custom CUDA streams, then we find that the average slowdown of task-0 is 10%, as shown by the H2D bars in Figure 5.14. Ideally, if the bidirectional bandwidth were to be twice the unidirectional bandwidth, then both the concurrent tasks would have no slowdown. In our experimental platform, however, the bidirectional bandwidth is only about 19.3% more than the unidirectional bandwidth according to the `simpleMultiCopy` CUDA SDK benchmark. Thus, task-0's slowdown is due to slower bidirectional bandwidth and not due to any possible MPI-ACC–related contention effects.

PCIe contention analysis with data transfers in the same direction: For this study, we analyze the contention impact of three `MPI_Send` implementations: MPI-ACC, manual pipelining using asynchronous MPI and CUDA, and manual synchronous MPI and CUDA. Since the Fermi GPUs have a single data transfer hardware queue in each direction, we expect significant contention when `MPI_Send` (task-0) is invoked concurrently with standalone D2H transfers on the same GPU (task-1). In fact, however, we show that MPI-ACC induces less contention than the manual synchronous and asynchronous MPI+GPU approaches of GPU data communication. We show that MPI-ACC enqueues commands to the GPU hardware queue in a balanced manner and minimizes the apparent performance slowdown.

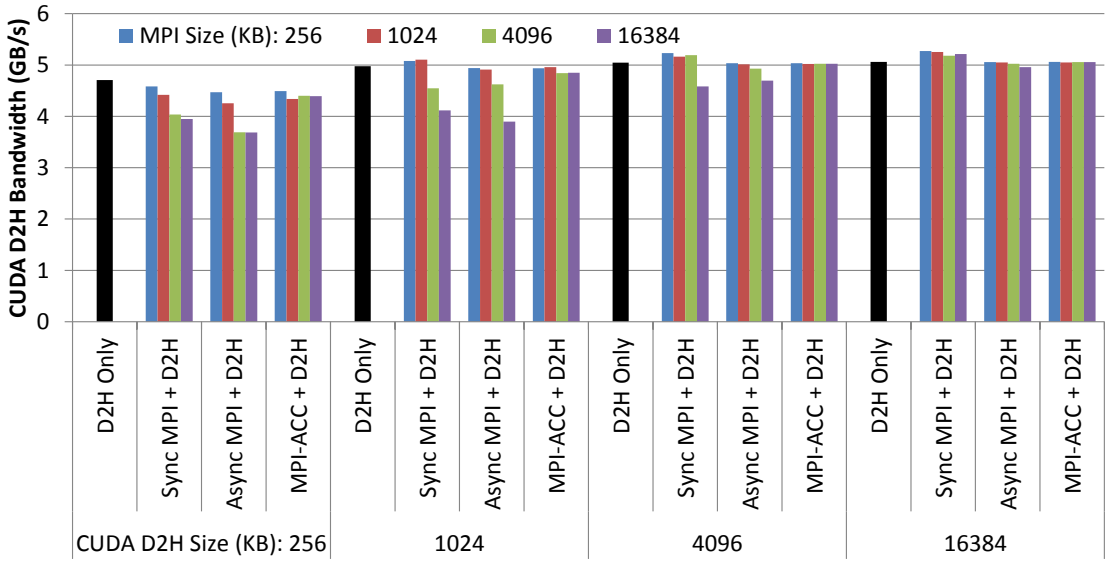
Figure 5.15a shows the slowdown of the MPI communication (task-0) for different combinations of data sizes across both tasks. In all three implementations, the slowdown in the MPI communication is minimal when the data sizes of both task-0 and task-1 are comparable. However, the slowdown increases as the relative data size of task-1 increases, because the D2H operations within the MPI communication will have to wait in the same hardware queue for the more time consuming D2H operations of task-1. On average, MPI-ACC is seen to have the least slowdown among the three MPI implementations. Furthermore, Figure 5.15b shows that MPI-ACC causes the least performance perturbation to the D2H operations of task-1, on average.

HPCToolkit analysis: We use HPCTOOLKIT's `Hpctraceviewer` interface to understand why MPI-ACC causes less contention than the manual MPI+GPU implementations do. `Hpctraceviewer` renders hierarchical, timeline-based visualizations of parallel hybrid CPU-GPU program executions. Figure 5.16 presents screenshots of the detailed execution profile of our contention benchmark. The `hpctraceviewer` tool presents the timeline information of all CPU processes, threads, and their corresponding CUDA streams. However, we zoom in and show only the timelines of the relevant CUDA streams associated with both tasks of the 0th process.

Figure 5.16a shows the effect of MPI-ACC's send operation interacting with the D2H data transfers of task-1. Since both tasks issue D2H commands and there is only one D2H queue on Fermi, we can see that only one of the CUDA streams is active at any given point in time. Moreover, the MPI-ACC's pipelining logic has been designed to not oversubscribe the GPU and leads to balanced execution, which can be seen by the interleaved bars in the MPI-related timeline. Figure 5.16b depicts the contention effect of the manual pipelined MPI+GPU implementation. In this implementation, we enqueue all the pipeline stages upfront, which is an acceptable design for standalone point-to-point communication. This design oversubscribes the GPU, however, and can be seen as clusters of bars in the MPI-related timeline. If one designs the manual MPI+GPU implementation similar to our MPI-ACC design, then the associated timeline figure will look like Figure 5.16a. The manual MPI+GPU implementation is more aggressive to enqueue GPU operations, and the D2H operations of task-1 tend to wait more. Figure 5.15b shows that, on average, MPI-ACC causes the least perturbation to the D2H task.

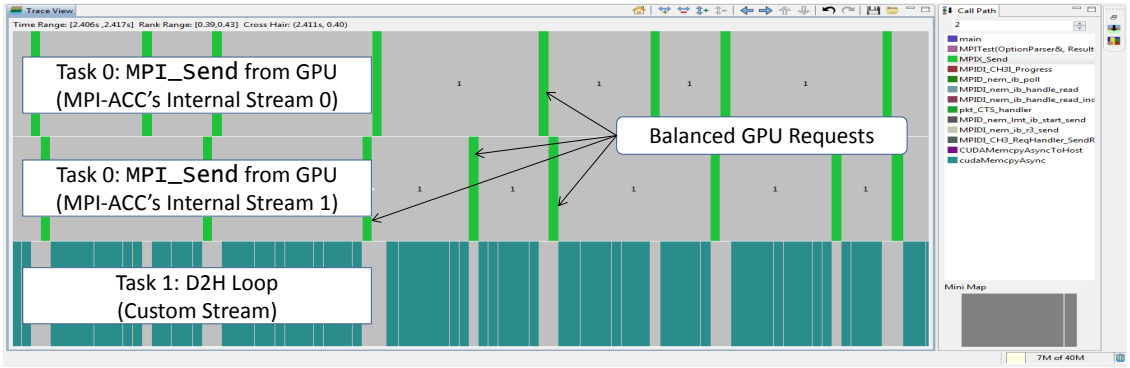


(a) Impact on the `MPI_Send` communication latency.

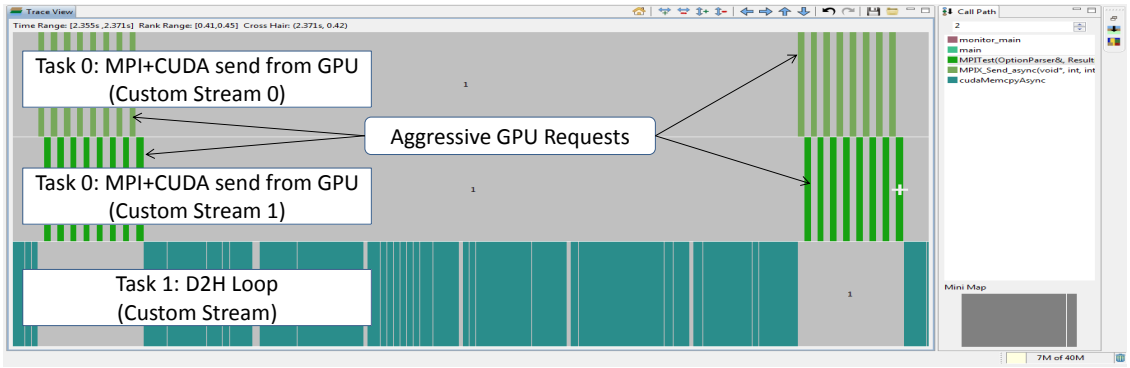


(b) Impact on the local device-to-host (D2H) GPU operations.

Figure 5.15: Characterizing the contention impacts of concurrent `MPI_Send` and local device-to-host (D2H) GPU operations.



(a) Impact of MPI-ACC's `MPI_Send` with concurrent D2H operations.



(b) Impact of manual MPI+GPU send task with concurrent D2H operations.

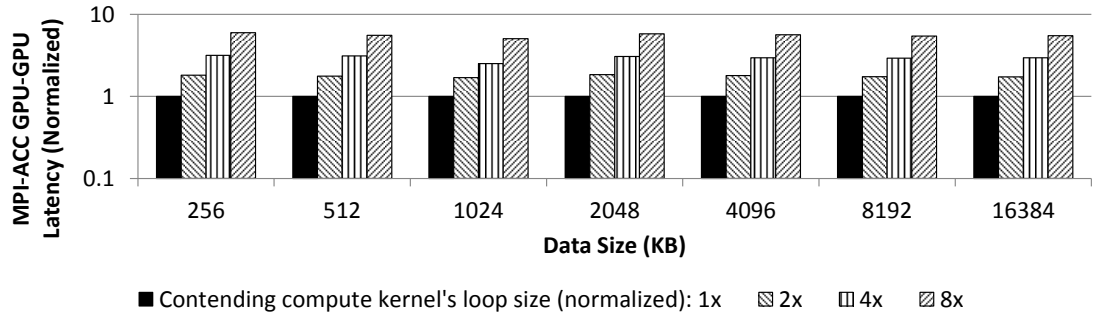
Figure 5.16: Using HPCTOOLKIT to understand the contention impacts of MPI-ACC and local GPU data transfer operations.

Quantifying the Software Contention

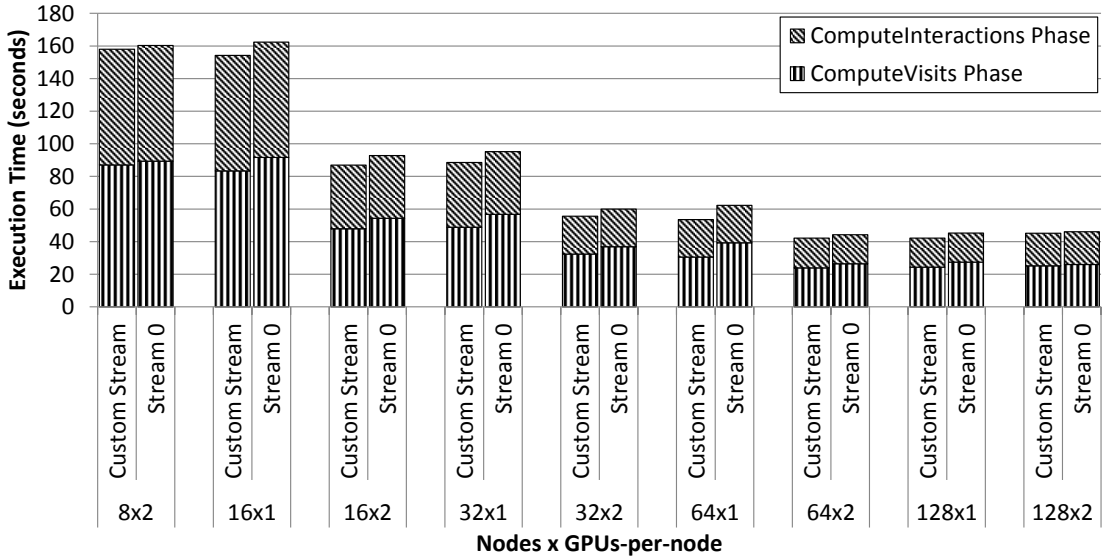
CUDA's stream-0 (default stream) is unique in that it is completely ordered with all operations issued on any stream of the device. That is, issuing operations on stream-0 would be functionally equivalent to synchronizing the entire device before and after each operation. Although MPI-ACC privately creates and uses custom streams to minimize software contention with other streams, a concurrent user operation to stream-0 can inadvertently stall any MPI-ACC operation on that GPU until stream-0 has completed.

In this section, we measure the impact of the stream-0 contention by using our benchmark as follows. We launch several MPI-ACC send-receives on task-0 and invoke several compute kernels on task-1, where the kernel simply executes a large multiply-add loop on the GPU. MPI-ACC uses custom (non-0) CUDA streams to schedule the pipelined data transfers in task-0 while the compute kernel of task-1 uses the CUDA stream-0. We vary the loop count on task-1 to artificially change the compute kernel's workload size and execution time. Figure 5.17a shows that as the loop size of the computation kernel task increases, the MPI task loses proportional performance. On the other hand, if task-1 uses the non-0 stream, task-0 and task-1 efficiently overlap, and the average contention impact drops to about 4% (Figure 5.14).

Contention due to stream-0 can be seen even in GPU-EpiSimdemics, and we analyze its effect as follows. In GPU-EpiSimdemics, the internode CPU-GPU communication of the visit messages is overlapped with a preprocessing kernel that performs data layout transformation (Section 5.3.1). While we use non-0 streams within MPI-ACC for the internode communication of visit messages, the preprocessing kernel may be launched with the user's chosen CUDA stream. Figure 5.17b shows that the performance of GPU-EpiSimdemics is about 6.6% slower when the preprocessing kernels use stream-0 instead of a non-0 stream, and the slowdown can be up to 16.3% for some node configurations. While MPI-ACC's streams are designed to scale, a poor application design using stream-0 can cause an apparent slowdown in MPI-ACC's data transfer performance.



(a) Impact on a point-to-point latency benchmark.



(b) Impact on the performance of GPU-EpiSimdemics.

Figure 5.17: Characterizing the contention impacts of CUDA's stream-0 on concurrent MPI operations.

5.5 Conclusion

In this chapter, we discussed MPI-ACC's comprehensive set of data transfer optimizations including data pipelining and buffer management. We studied the efficacy of MPI-ACC for scientific applications from the domains of epidemiology (GPU-EpiSimdemics) and seismology (FDM-Seismology) and we presented the lessons learned and tradeoffs. We found that while MPI-ACC's internal pipeline optimization helped improve the end-to-end communication performance, it enabled novel optimization opportunities at the application level which significantly enhanced the CPU-GPU and network utilization. With MPI-ACC, one could naturally express the communication target without explicitly treating the CPUs as communication relays. MPI-ACC decoupled the application logic from the low-level GPU communication optimizations, thereby significantly improving scalability and application portability across multiple GPU platforms and generations. We also provided insights into the scalable design of MPI-ACC. Specifically, we showed that MPI-ACC delivers maximum concurrency by carefully ordering multiple GPU streams and efficiently balancing the host-to-device (H2D) and device-to-host (D2H) hardware queues for data pipelining.

Chapter 6

Task Mapping with MPI-ACC

6.1 Introduction

Each MPI process in the hybrid model consists of a host component that is run by the OS on the CPU core(s) and a device component that is run by CUDA/OpenCL on the local device(s). The host process is automatically assigned to the appropriate CPU core by the OS scheduler, and additional tools/libraries like `numactl`/`libnuma` can be used to guide the process-CPU mapping. However, the current GPU programming models require the programmers to explicitly choose the device for kernel offloading. With increasing heterogeneity within a node, it is critical for a runtime system to assign the optimal GPU for a given kernel. Our experiments and other work in the literature [30] indicate that the peak performance ratios of the GPUs do not always translate to the optimal 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. In this chapter, we address the task-device mapping issue as follows:

- Microbenchmark-based performance analysis for memory-bound GPU kernels, which is based on [12], in Section 6.2.

- Device selection strategy that leverages our generic performance projection technique, which is based on [16], in Section 6.3.
- Design and implementation of our runtime system component of MPI-ACC's task mapping subsystem, which we call as MultiCL [17], in Section 6.4.

6.2 Memory Modeling Example

In this section, we describe our methodology of using offline microbenchmarks and statistical regression techniques to analyze memory-bound GPU kernels. In the next section, we leverage our lessons learned and apply the same methodology to understand the kernels' performance limiters (compute, on-chip or off-chip memory), which we then use to pick the optimal device for the given kernel.

6.2.1 Partition Camping in Memory-bound GPU Kernels

We choose *partition camping* as the example scenario and predict a performance range for memory-bound GPU kernels. The performance bound helps us to analyze the degree to which partition camping exists in the kernel, which can lead to memory-access optimizations. Partition camping is caused by kernel-wide memory accesses that are skewed towards a subset of the available memory partitions or banks, which may severely affect the performance of GPU kernels [64]. It must be noted that the impact due to partition camping has reduced since the NVIDIA Fermi architecture, but is severe for the pre-Fermi GPUs. Our study shows that the performance can degrade by up to seven-fold because of partition camping (Figure 6.1). Common optimization techniques for NVIDIA GPUs have been widely studied, and many tools and models are available to perform common intra-block optimizations. It is difficult to discover and characterize the effect of partition camping, because the accessed memory addresses and the actual time of memory transactions have to be analyzed together. Therefore, traditional methods that detect similar problems, such as static code analysis techniques to discover shared-memory bank conflicts or the approaches used in existing GPU performance models, are prone to errors because they do not analyze the timing information.

6.2.2 Predicting Performance Bounds

We develop a suite of microbenchmarks to capture the device characteristics. We run the microbenchmarks offline and record the effect of various memory-access patterns combined with the different memory-transaction types and sizes. We then use the actual kernel's configuration and memory-access patterns and match it with the device characteristics to predict performance. For this specific example, we obtained the number of global memory transactions, their type and sizes from performance counters via the NVIDIA Profiler.

For the partition-camping problem, we predict a range of possible execution times, which denotes the degree to which partition camping can exist in the kernel. While partition camping truly means that any subset of memory partitions are being accessed concurrently, we choose the extreme cases for our study, i.e. all the available partitions are accessed uniformly (Without Partition Camping, or Without PC, for short), or only one memory partition is accessed all the time (With Partition Camping, or With PC, for short). Although this method does not exhaustively test the different degrees of partition camping, our study acts as a realistic first-order approximation to characterize its effect in GPU kernels. Thus, we developed two sets of benchmarks and analyzed the memory effects with and without partition camping. Each set of benchmarks test the different memory-transaction types (reads and writes) and different memory-transaction sizes (32, 64 and 128 bytes), making it a grand total of twelve benchmarks. As an example, we show in Figure 6.1 that the performance of memory-bound kernels can degrade by up to seven-fold if kernels suffer from partition camping. This particular result was obtained by running a simple 64-byte memory read micro-kernel that was part of our micro-benchmark suite.

We apply multiple linear-regression techniques on the microbenchmark data and predict the the performance of the given kernels by extracting their memory-access characteristics. Our performance model predicts a range of the effect of partition camping in a GPU kernel. If performance is measured by the wall clock time, the lower bound of our predicted performance will refer to the best case, i.e. without partition camping for any memory transaction. The upper bound will refer to the worst case, i.e. with partition camping for all memory transaction types and sizes. Figure 6.2 shows that our approach and performance model are accurate for the particular molecular modeling application. As

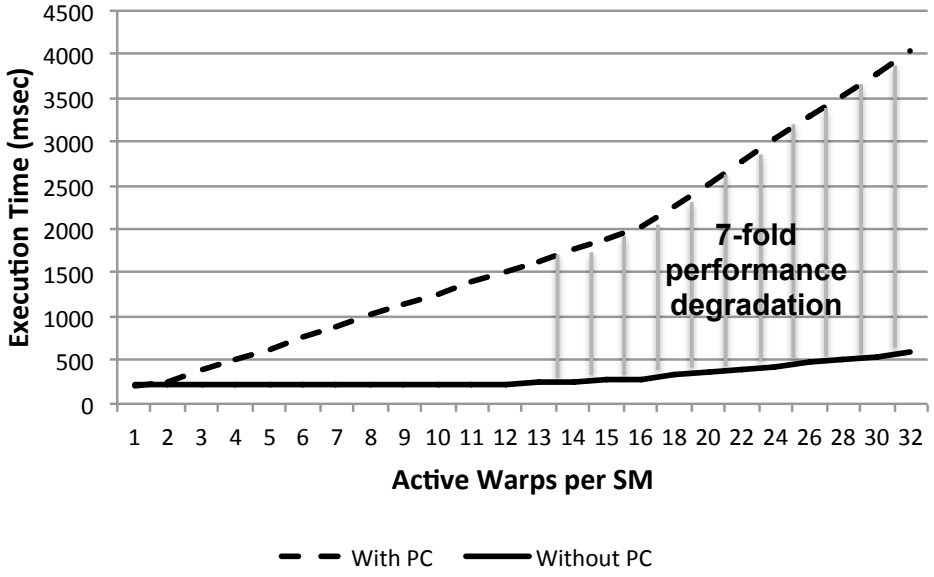


Figure 6.1: The negative effect of partition camping (PC) in GPU kernels

a research artifact, we developed a simple spreadsheet-based tool called *CampProf*, which helps the user to visually detect and analyze the partition camping effects in the GPU kernel (Figure 6.3).

6.2.3 Lessons Learned

We outline below the lessons learned from using microbenchmarks for analyzing memory-bound kernels.

- While microbenchmarks are useful to create offline device profiles, performance counters and tools like the NVIDIA Profiler can be used to obtain online kernel profiles.
- For a runtime system, obtaining the kernel characteristics via performance counters is impractical for for device selection because (1) the device may be on a remote node which would incur network costs, and (2) the device may be busy with some other kernel.

While we still use the microbenchmark-based modeling approach for device selection, we will show that we use functional emulators to obtain the kernel characteristics like instruction mix, memory accesses and cache statistics.

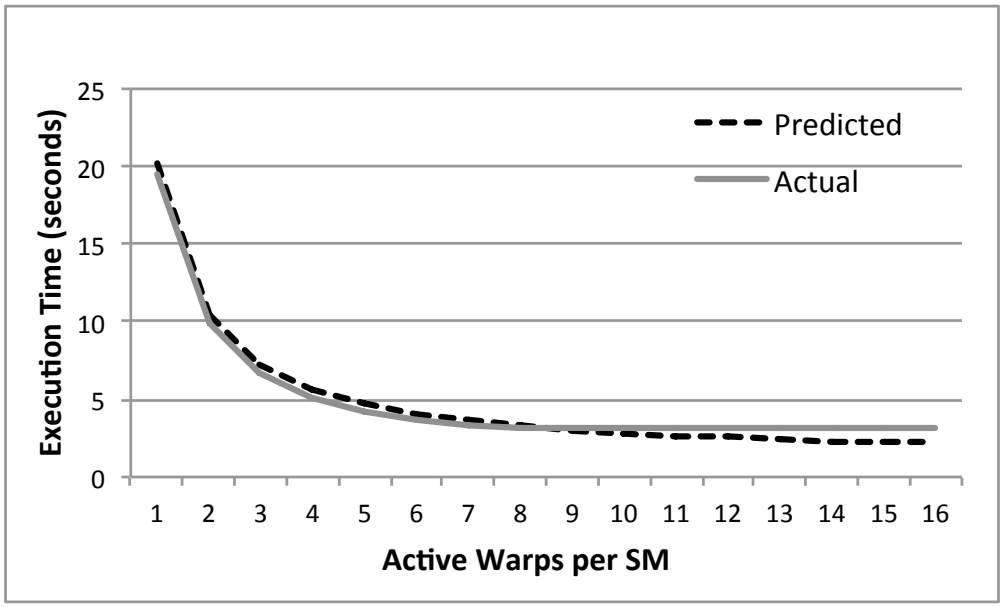


Figure 6.2: Validating the performance prediction model for a molecular modeling application.

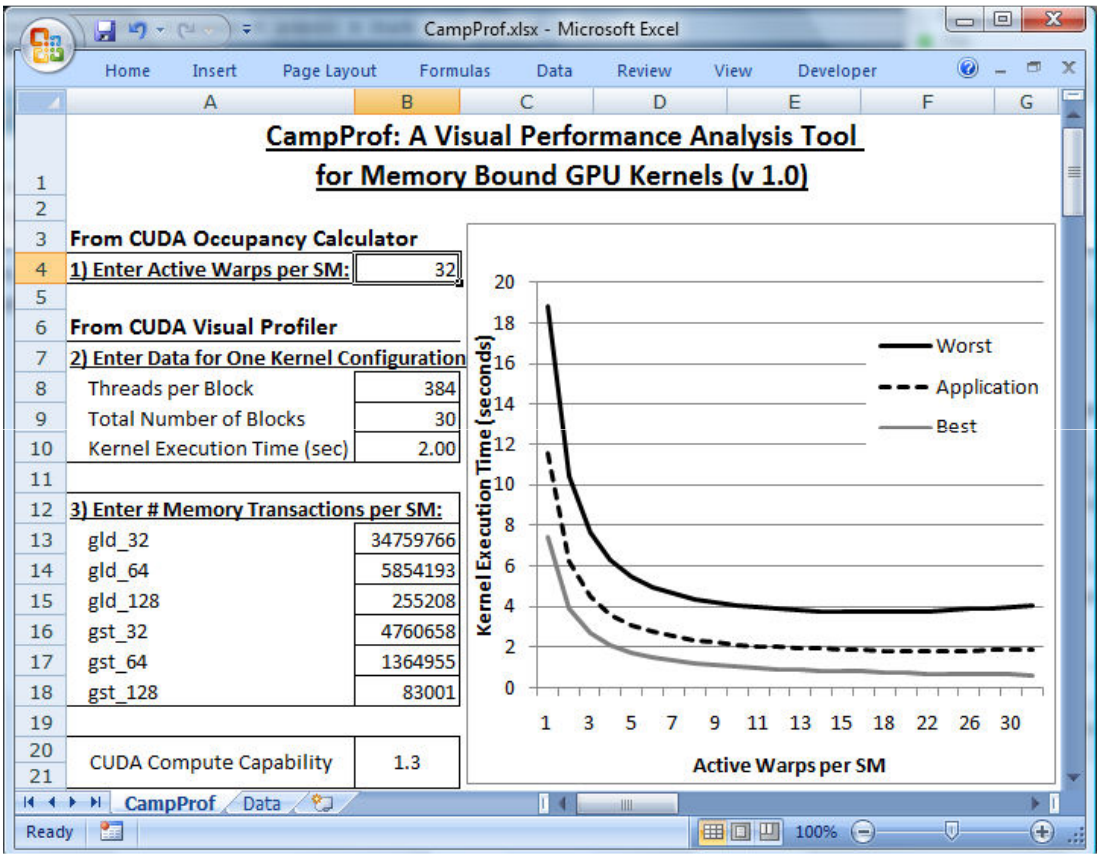


Figure 6.3: Screenshot of the CampProf tool.

6.3 Device Selection Strategies

The goal of online performance projection is twofold: (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 [21] and Multi2Sim [76], 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 [33] 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 larger. On the other hand, the overhead will be much smaller than with cycle-accurate emulation.

6.3.1 Approach

We realize a variant of the dynamic kernel analysis for performance projection while significantly limiting the emulation overhead with acceptable 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 6.4 shows, our online projection consists of three steps: static profiling, dynamic profiling, and performance projection. First, we obtain the hardware characteristics through offline profiling (or static profiling). 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.

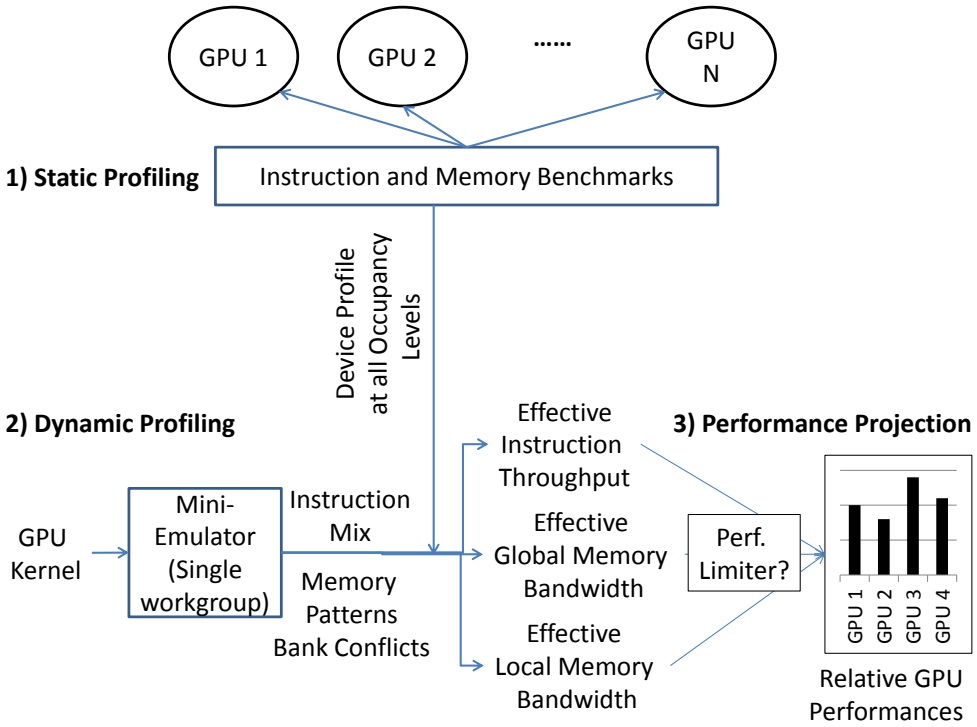


Figure 6.4: The performance projection methodology.

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 workgroup of the kernel. With such an approach, we can ob-

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

tain 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-workgroup characteristics and the per-device hardware profile, we project 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 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.

6.3.2 Offline Device Characterization

Our device characterization focuses on three major components of GPU performance: instruction throughput, local memory throughput, and global memory throughput. The instruction 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 offline profiling. We use microbenchmarks derived from the SHOC benchmark suite [31] 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 6.5 shows the coalesced memory throughput behavior for the AMD Radeon 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 6.3.4.

Similar to global memory, the local memory benchmarks measure the throughput of local memory at varying occupancy levels of GPU. 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

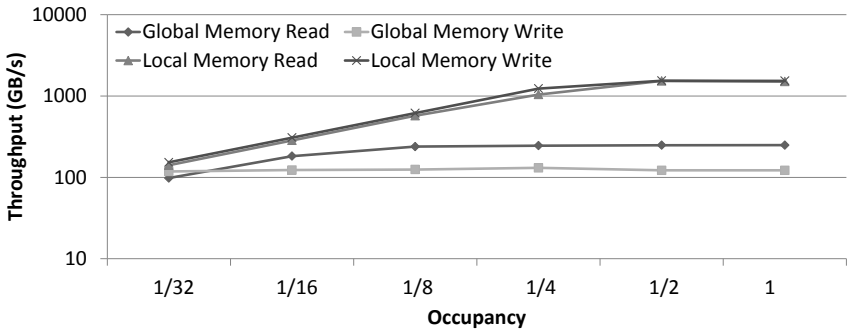


Figure 6.5: Memory throughput on the AMD Radeon HD 7970.

performance as described in Section 6.3.4.

6.3.3 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, intensity of branch divergence, 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 [18]. 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 problem of emulation overhead, we propose a technique named mini-emulation, which employs a modified emulator that functionally emulates just a single workgroup when invoked. Our assumption is that workgroups 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 workgroups and the statistics of a single workgroup, thereby significantly reducing the emulation overhead.

To emulate both NVIDIA and AMD devices, we adopt and modify two third-party emulators: GPGPU-Sim [21] for NVIDIA devices and Multi2sim [76] 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 workgroup, ignores the remaining workgroups and outputs the appropriate statistics. We do not change the task assignment logic in the emulator, i.e., the single emulated workgroup still performs the same amount of work as if it were part of the full kernel having many workgroups.

Deriving Full Kernel Characteristics

The mini-emulator generates characteristics of only a single workgroup. To cast performance projections, we need to obtain the characteristics of the full kernel. The scaling factor between the characteristics of a single workgroup 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 workgroup to that of the full kernel. The derived characteristics of the full kernel can then be used to project the kernel's performance.

6.3.4 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 [84] 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 threads) can generate up to 32 128-byte 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 GPU. 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. $max(t_{compute}, t_{global}, t_{local})$.

6.3.5 Lessons Learned

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

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 ran 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 6.1: Summary of GPU devices.

GPU	Architecture Name	Compute Units	Peak Performance (GFlops)	Peak Memory Bandwidth (GB/s)	Memory Transaction Sizes (B)	Shared Memory 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 6.2: Summary of applications.

Floyd Warshall	Fast Walsh Transform	Matrix Multiply (global memory)	Matrix Multiply (local memory)
Nodes = 192	Size = 1048576	Size = [1024,1024]	Size = [1024,1024]

Reduction	NBody	AES Encrypt-Decrypt	Matrix Transpose
Size = 1048576	Particles = 32768	W=1536, H=512	Size = [1024,1024]

Table 6.1 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 6.2 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 6.3.4, to identify the performance bottlenecks in computation, global memory throughput, and local memory throughput. Figure 6.6 presents the projected normalized execution times for the AMD and NVIDIA devices and characterizes the applications by the performance limiting components.

Figure 6.6 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,

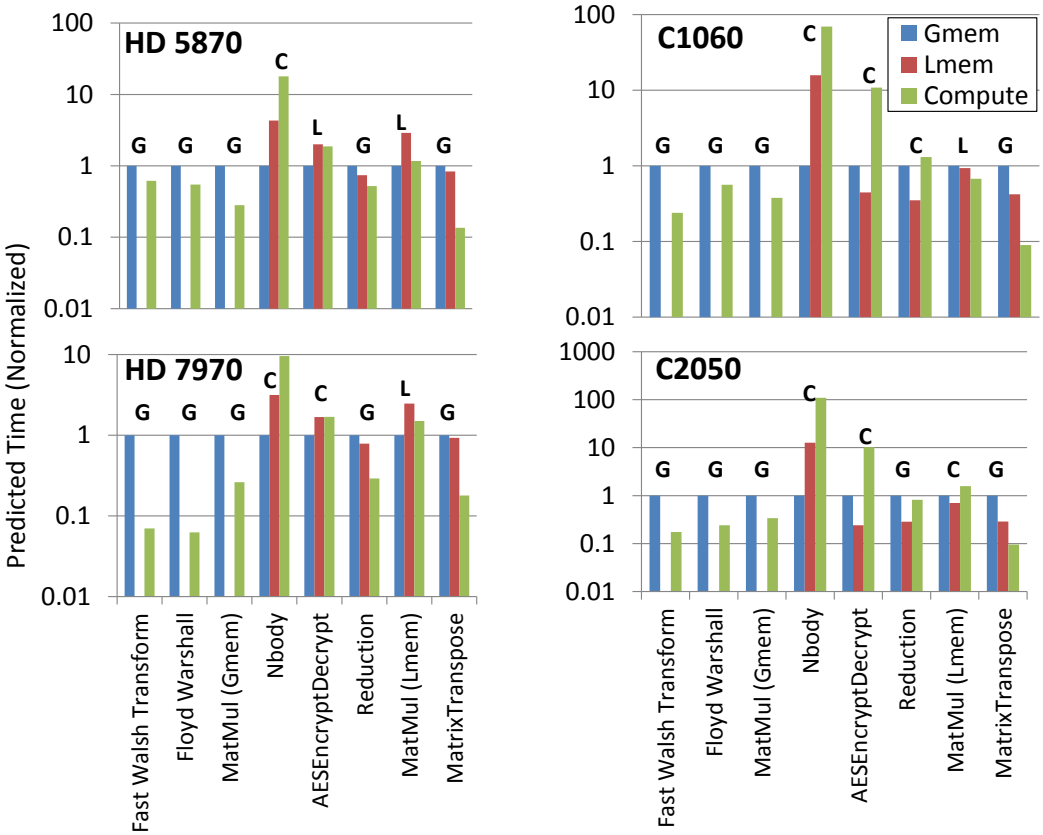
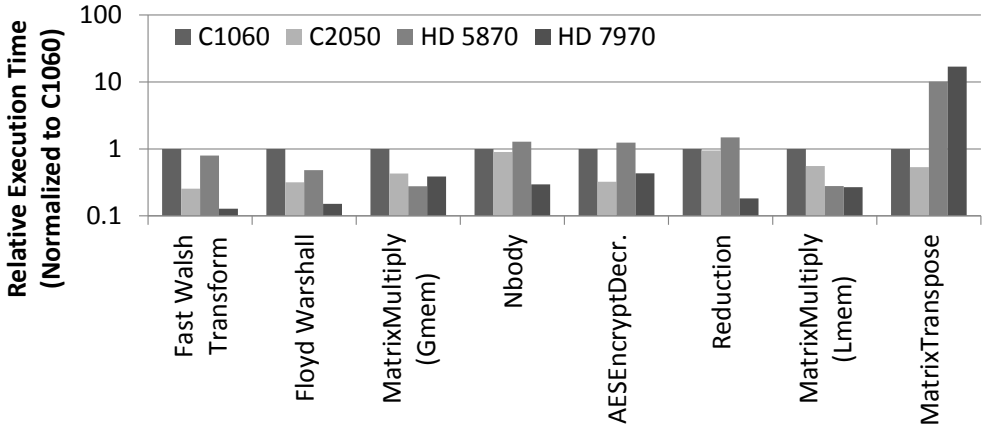
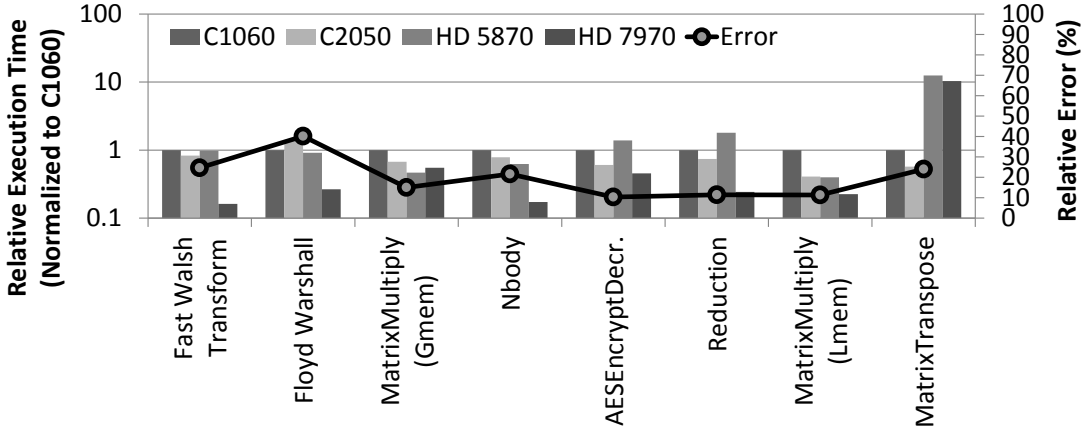


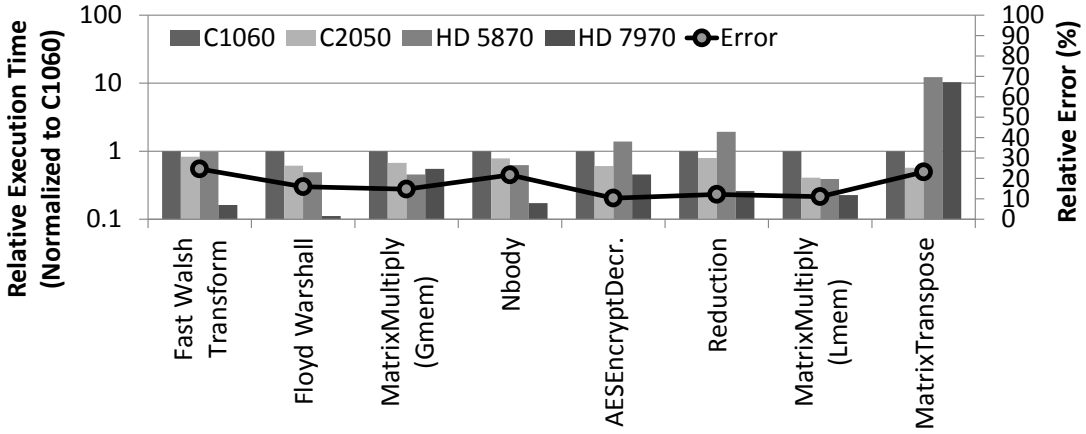
Figure 6.6: Analysis of the performance limiting factor. Gmem stands for global memory and Lmem stands for local memory. MatMul (Gmem) stands for the matrix multiplication benchmark that only uses the GPU's global memory and MatMul (Lmem) stands for the matrix multiplication benchmark with the local memory optimizations. The performance limiter of an application is denoted at the top of each bar: G for Gmem, L for Lmem and C for Compute.



(a) Actual execution times.



(b) Projected execution times by single workgroup mini-emulation.



(c) Projected execution times by full kernel emulation.

Figure 6.7: Accuracy of the performance projection model. Gmem stands for global memory and Lmem stands for local memory.

while the boundedness of other applications changes across some of the devices. Figure 6.6 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 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.

Analysis

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 6.7a shows the actual execution time of all benchmarks on our four test GPUs, and Figures 6.7b and 6.7c show the projected execution times by the single workgroup mini-emulation and the full kernel emulation, respectively. All the numbers are normalized to the performance of NVIDIA C1060.

Optimal Device Selection: 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 over the recommended GPU. Figure 6.7 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 Radeon HD 7970, whereas the best device is C2050, with an optimal 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 may need information

Table 6.3: Performance model overhead reduction – ratio of full-kernel emulation time to single workgroup mini-emulation time.

Application	Fast Walsh Transform	Floyd Warshall	Matrix Multiply (Global Memory)	Nbody	AES Encrypt	Reduction	Matrix Multiply (Local Memory)	Matrix Transpose
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

about the kernel's relative performance among the GPUs in addition to the optimal GPU for each kernel. This helps the runtime 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 workgroup 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 workgroup 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 workgroups of the kernel must be independent of each other and that all the workgroups 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 workgroup 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

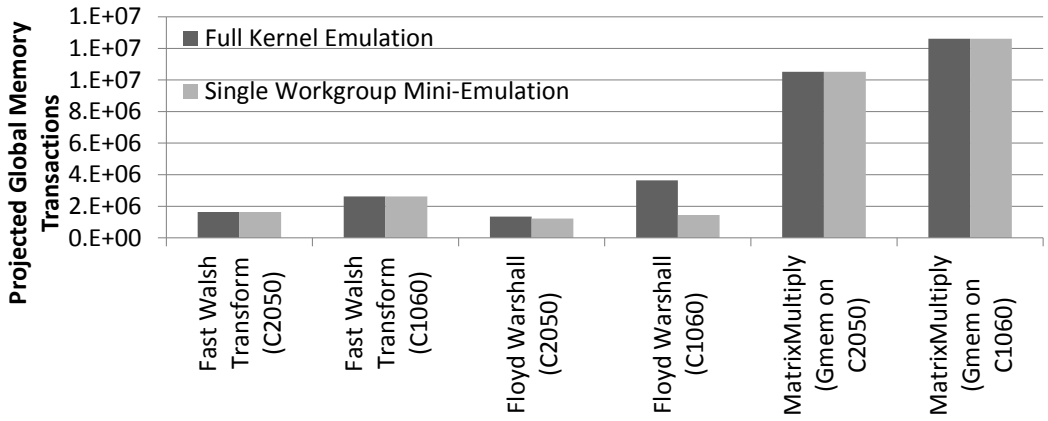


Figure 6.8: Global memory (Gmem) transactions for select applications.

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 (Figure 6.8). 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 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 offline, 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 workgroup to collect kernel statistics.

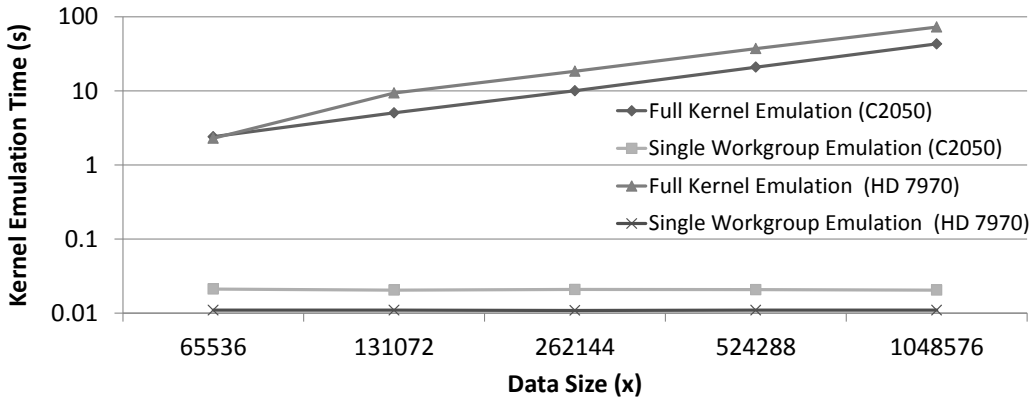
The state-of-the-art technique to obtain detailed runtime statistics of a kernel is full kernel emulation. As Table 6.3 shows, our mini-emulation approach reduces the emulation overhead by orders of magnitude. Meanwhile, it obtains the same level of details about runtime characteristics. In fact, the 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 data set, 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 data sets; 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.

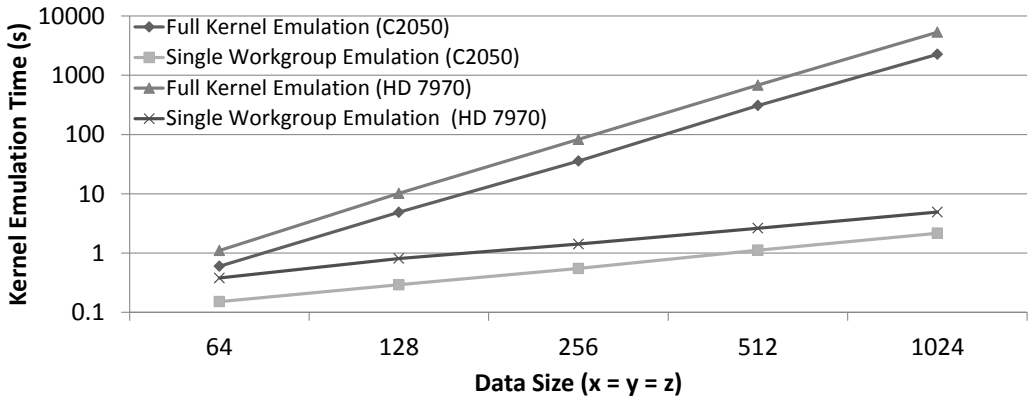
Impact of Input Data Sizes: Figure 6.9 shows the performance impact of the input data size on the full-kernel emulation and single workgroup mini-emulation overheads. Figure 6.9a shows that the full-kernel emulation overhead for the reduction kernel increases with data size. The reduction kernel launches as many workitems as the data elements, thereby having a one-to-one mapping between the workitem and the data element. As the data size grows, the number of workitems also increases, so that each workitem and workgroup has a constant amount of computation. Since each workitem 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 workgroup 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 6.9b 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 workitems as the output matrix size, but unlike the reduction kernel, each workitem and workgroup do not have a constant amount of computation. The load on each workitem 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 6.10 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 3GB global



(a) Kernel: Reduction.



(b) Kernel: Matrix Multiplication (using local memory).

Figure 6.9: Kernel emulation overhead – full kernel emulation vs. single workgroup mini-emulation.

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 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.

6.4 Design and Implementation of the Task Mapping Runtime

In this section we explain the design of the MultiCL runtime system for optimal task mapping and discuss key optimization tradeoffs and our evaluation. Specifically, we explain the overhead vs. optimality tradeoff between the static

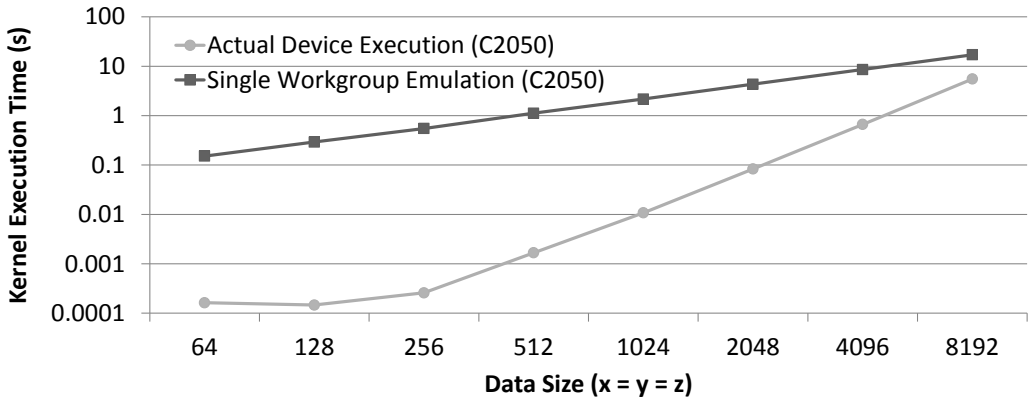


Figure 6.10: Kernel emulation overhead – single workgroup mini-emulation vs. actual device execution.

and dynamic command-queue scheduling strategies. For dynamic device selection, we describe techniques for overhead reduction, such as efficient device-to-device data movement for data-intensive kernels, mini-kernel profiling for compute-intensive kernels, and data caching and data reuse for future kernel invocations.

6.4.1 The SnuCL Runtime Framework

SnuCL [46] is a cross-platform OpenCL implementation that provides the programmer with a single platform view of a disparate set with OpenCL devices, where the OpenCL devices can be from multiple vendor platforms. SnuCL provides two programming modes to the end user: (1) single mode, and (2) cluster mode. The single mode provides the standard OpenCL programming interface to the programmer, with the added functionality of unifying all the different vendor platforms and devices under a single "SnuCL" platform. Programmers write OpenCL code as before, but can also share data, share kernels and synchronize across devices from all supported vendor platforms. In the SnuCL cluster mode programming model, there is one host node and multiple backend nodes. SnuCL enables the application to use all the OpenCL devices in the host and backend nodes in the cluster as if they were on the host node itself. The programmer writes OpenCL programs for a single process or address space, and the SnuCL cluster mode uses MPI as the underlying communication library to perform data resolution across the backend nodes and devices in the system. We focus on the single mode of SnuCL, where the programmer writes MPI+OpenCL programs, where the OpenCL code is meant to be executed within a single node but across all available devices within the node. The programmer has

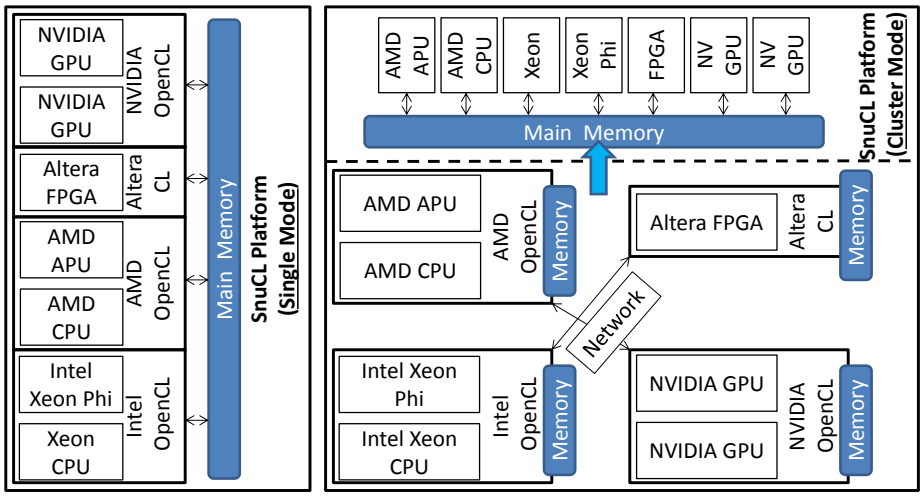


Figure 6.11: Left: SnuCL's 'single' mode. All OpenCL platforms within a node are aggregated under a single platform. Right: SnuCL's 'cluster' mode. All OpenCL devices and platforms across a cluster of nodes are a provided with the view of a single OS image with a shared resource abstraction. SnuCL helps in sharing data, sharing kernels and synchronization across devices from different platforms.

to manually choose the OpenCL device for each kernel and data transfer operation, i.e. there is no automatic device selection. Figure 6.11 shows the different programming abstractions provided by SnuCL.

6.4.2 Designing the MultiCL Runtime System

The SnuCL runtime creates a *scheduler* thread per user process, but the default scheduler thread statically maps the incoming commands to the explicitly chosen target device—that is, manual scheduling. MultiCL is our extension of the SnuCL runtime, with the added functionality of automatic command-queue scheduling support to OpenCL programs. MultiCL's design is depicted in the left portion of Figure 6.12. The user's command queues that are created with the `SCHED_OFF` flag will be statically mapped to the chosen device, whereas those that have the `SCHED_AUTO` flag are automatically scheduled by MultiCL. Further, the user-specified context property (e.g.: `AUTO_FIT`) determines the scheduling algorithm for the pool of dynamically mapped command queues. Once a user queue is mapped to the device, its commands are issued to the respective device specific queue for final execution.

The MultiCL runtime consists of three modules: (1) device profiler, where the execution capabilities (memory, com-

pute and I/O) of the participating devices are collected or inferred; (2) kernel profiler, where kernels are transformed and their execution times on different devices are measured or projected; and (3) device mapper, where the participating command queues are scheduled to devices so that queue completion times are minimal. The OpenCL functions that trigger the respective modules are shown in the right portion of Figure 6.12.

Device Profiler

The device profiler, which is invoked once during the `clGetPlatformIds` call, retrieves the static device profile from the profile cache. If the profile cache does not exist, then the runtime runs data bandwidth and instruction throughput benchmarks and caches the measured metrics as static per-device profiles in the user's file system. The profile cache location can be controlled via environment variables. The benchmarks are derived from the SHOC benchmark suite [31] and NVIDIA SDK, and are run for a wide range of data sizes ranging from being latency bound to bandwidth bound. Benchmarks measuring host-to-device (H2D) bandwidths are run for all the CPU socket–device combinations, whereas the device-to-device (D2D) bandwidth benchmarks are run for all device–device combinations. These benchmarks are included as part of the MultiCL runtime. Bandwidth numbers for unknown data sizes are computed using simple interpolation techniques. The instruction throughput of a device (or peak flop rate) can also be obtained from hardware specifications and manually included in the device profile cache. The benchmarks are run again only if the system configuration changes, for example, if devices are added or removed from the system or the device profile cache location changes. However, in practice, the runtime just reads the device profiles from the profile cache once at the beginning of the program.

Kernel Profiler

Kernel execution times can be estimated by performance modeling or performance projection techniques, but these approaches are either done offline or they are impractical because of their large runtime overheads. We follow a more practical approach in that we run the kernels once per device and store the corresponding execution times as part of the kernel profile. While this approach may cause potential runtime overhead to the current programs, we

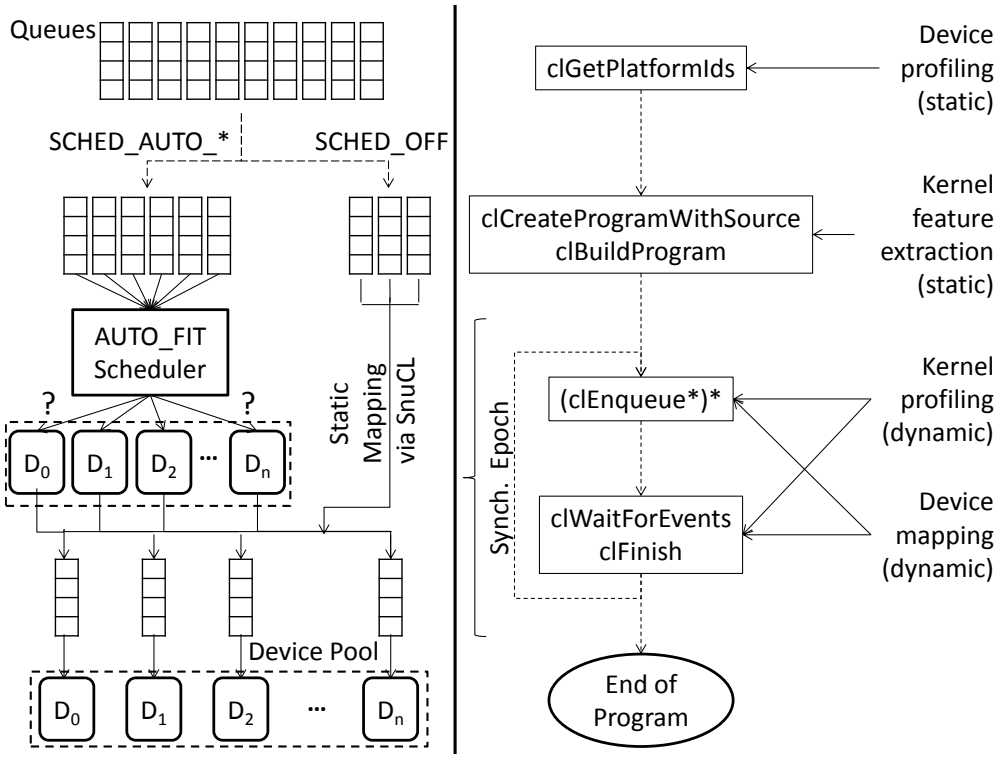


Figure 6.12: Left: MultiCL runtime design and extensions to SnuCL. Right: invoking MultiCL runtime modules in OpenCL programs.

discuss several ways to mitigate the overhead in this section. Also, our experiments indicate that, upon applying the runtime optimizations, the runtime overhead is minimal or sometimes negligible while the optimal device combinations are chosen for the given kernels. Static kernel transformations, like minikernel creation, is performed during `clCreateProgramWithSource` and `clBuildProgram`, whereas dynamic kernel profiling is done at synchronization points or at user-defined code regions.

Device Mapper

Each `clEnqueue-` command is intercepted by the device mapper and the associated queue is added to a ready queue pool for scheduling. Once the scheduler is invoked and maps the queue to a device, the queue is removed from the queue pool. On the one hand, the scheduler can *actively* be invoked for every kernel invocation, but that approach can cause significant runtime overhead due to potential cross-device data migration. On the other hand, the runtime can simply aggregate the profiled execution costs for every enqueue command, and the scheduler be invoked at synchronization epoch boundaries or at any other user-specified location in the program. The scheduler options discussed in the previous section can be used to control the frequency and location of invoking the scheduler, which can further control the overhead vs. optimality tradeoff.

6.4.3 Static Command-Queue Scheduling

Users can control which command queues participate in static queue scheduling (`SCHED_AUTO_STATIC`) and which of them are scheduled dynamically (`SCHED_AUTO_DYNAMIC`). In the static command-queue scheduling approach, we use the device profiler and device mapper modules of our runtime and do not perform dynamic kernel profiling; in other words, we statically decide the command-queue schedules based only on the device profiles. Users can select this mode as an approximation to reduce scheduling overhead, but the optimal device may not be selected at certain times. The MutliCL runtime uses the command queue properties (compute intensive, memory intensive, or I/O intensive) as the selection criterion and chooses the best available device for the given command queue.

6.4.4 Dynamic Command-Queue Scheduling

In the dynamic command-queue scheduling approach, we use the kernel profiling and device mapping modules of our runtime and selectively use the device profiling data. That is, we dynamically decide the command-queue schedules based only on the kernel and device profiles.

Reducing Overhead in Kernel Profiling

While dynamic kernel profiling performs more accurate task-device mapping, it incurs non-negligible profiling overhead. Users can choose runtime options to mitigate the runtime overhead associated with dynamic kernel profiling. Next, we discuss three techniques to reduce the kernel profiling overhead.

Kernel Profile Caching for Iterative Kernels We cache the kernel profiles in memory as key-value pairs, where the key is the kernel name and the value is its performance vector on the devices. The cached kernel profiles are used to schedule future kernel invocations, and our approach significantly reduces kernel profiling overhead. We define a *kernel epoch* as a collection of kernels that have been asynchronously enqueued to a command queue. Synchronizing after a kernel epoch on the command queue will block until all the kernels in the epoch have completed execution. We also cache the performance profiles of kernel epochs for further overhead reduction. The key for a kernel epoch is just the set of the participating kernel names, and the value is the aggregate performance vector of the epoch on all the devices. The user can provide runtime options to either batch schedule kernel epochs or individual kernels.

Iterative kernels benefit the most from the kernel profile cache because of kernel reuse. However, some kernels may perform differently across iterations, or their performances may change periodically depending on the specific phase in the program. The user can set a program environment flag to denote the iterative scheduler frequency, which tells our scheduler when to recompute the kernel profiles and rebuild the profile cache. In practice, we have found iterative kernels to have the least overhead, because the overhead that is incurred during the first iteration or a subset of iterations is amortized over the remaining iterations.

Mini-kernel Profiling for Compute-Intensive Kernels While the kernel profile caching helps iterative applications, noniterative applications still incur profiling overhead, and this is especially true for compute intensive kernels. To choose the best device, we need to know only the *relative* kernel performances and not necessarily the absolute kernel performances. Therefore, we create a profiling and modeling technique called minikernel profiling, where we run just a single workgroup of the kernel on each participating device and collect the relative performances in the kernel profiles. Our approach dramatically reduces runtime overhead, as discussed in Section 6.5. Our assumption is that workgroups often exhibit similar behavior and share similar runtime statistics, a situation typical of data-parallel workloads.

Creating the mini-kernel source and compiling the binary object: One approach for running a single workgroup is simply to launch the kernel with a single workgroup configuration: global work size equals the local work size. However, some kernels assign work to each workgroup proportional to the problem size, whereas other kernels assign constant work to each workgroup but the number of workgroups is proportional to the problem size. If we launch a kernel with a single workgroup, the work distribution logic in the kernel may make the workgroup work on the entire problem size, and thus it does not guarantee a reduction in the profiling overhead. To solve this problem, we create a new *mini-kernel* for every kernel in the program and store the mini-kernel source in memory. The mini-kernel is created by modifying the source kernel logic to create a conditional that allows just the first workgroup to execute the kernel and force all the other workgroups to return immediately (e.g., Figure 6.13). We launch the mini-kernel with the same launch configuration as the original kernel, so the amount of work done by the first workgroup in the mini-kernel does not change. Our approach thus guarantees reduction in the profiling overhead.

The mini-kernel profiling approach is conceptually similar to the mini-emulation technique that we explored previously [58]. The purpose of mini-emulation is to indirectly compute the relative performance of kernels by collecting kernel characteristics from modified emulators, whereas the purpose of mini-kernel profiling is to directly collect the relative performance of kernels by running single workgroups on available devices. Also, mini-emulation modifies a few emulators to functionally emulate a single workgroup, whereas mini-kernel profiling involves quick on-the-fly string substitutions in the kernel source itself to create a conditional statement that forces just a single workgroup to execute through the kernel.

```
1  // Actual kernel code
2  __kernel void foo(...) {
3    /* kernel code */
4  }
5
6  // Mini-kernel code
7  __kernel void foo(...) {
8    /* return if this is not first workgroup */
9    if(get_group_id(0)+get_group_id(1)+get_group_id(2)!=0)
10     return;
11   /* kernel code */
12 }
```

Figure 6.13: Mini-kernel example.

We intercept the `clCreateProgramWithSource` call and create the mini-kernel by simple string manipulation techniques. We intercept the `clBuildProgram` call and build the program with the new mini-kernels into a separate binary. We store each mini-kernel object as a shadow kernel within the original kernel object in our runtime and invoke them during kernel profiling. While this method doubles the cost of building the OpenCL source at runtime, we consider this to be initial setup cost that does not change the actual runtime of the program. We note also that the mini-kernel profiling approach requires access to the kernel source to perform the above optimization.

Data Caching for I/O-Intensive Kernels One of the steps in kernel profiling is to transfer the input data sets from the source device to each participating device before running the mini-kernels on them. Clearly, the data transfer cost adds to the runtime overhead. If there are n devices, the brute-force approach is to do device-to-device (D2D) data transfers $n-1$ times from the source device to every other device, followed by an intra-device data transfer at the source. However, the current vendor drivers do not support direct D2D transfer capabilities across vendors and device types. Thus, each D2D transfer is performed as a device-to-host (D2H) and host-to-device (H2D) double operation via the host memory, which means that there will be $n-1$ D2H and $n-1$ H2D operations[2]. However, we realize that the host memory is shared among all the devices within a node. Therefore, we optimize the data transfer step by doing just a single D2H copy from the source device to the host, followed by $n-1$ H2D data transfers. In addition, we cache the incoming data sets in each destination device so that if our runtime mapper decides to migrate the kernel to a different target device, the required data is already present in the device. With this optimization, however, we trade

[2]GPUDirect for NVIDIA GPUs has very limited OpenCL support.

off increased memory footprint in each device for less data-transfer overhead.

6.5 Evaluation

We describe the experimental setup and demonstrate the efficacy of our runtime optimizations using a benchmark suite and a real-world seismology simulation application.

Our experimental node has a dual-socket oct-core AMD Opteron 6134 (Magny-Cours family) processor and two NVIDIA Tesla C2050 GPUs. Each CPU node has 32 GB of main memory, and each GPU has 3 GB of device memory. We use the CUDA driver v313.30 to manage the NVIDIA GPUs, and the AMD APP SDK v2.8 to drive the AMD CPU OpenCL device. We compile our programs using GCC v4.6.3 on the Linux kernel v3.2.35. The network interface is close to CPU socket 0 and the two NVIDIA GPUs have affinity to socket 1, which creates non-uniform host-device and device-device distances (and therefore data transfer latencies) depending on the core affinity of the host thread. The MultiCL runtime scheduler incorporates the heterogeneity in compute capabilities as well as device distances when making device mapping decisions.

6.5.1 NAS Parallel Benchmarks (NPB)

The NAS Parallel Benchmarks (NPB) [20] are designed to help evaluate current and future parallel supercomputers. The SnuCL team recently developed the SNU-NPB suite [68], which consists of the NPB benchmarks ported to OpenCL. The SNU-NPB suite also has a multidevice version of the OpenCL code (SNU-NPB-MD) to evaluate OpenCL's scalability. SNU-NPB-MD consists of six applications: BT, CG, EP, FT, MG, and SP. The OpenCL code is derived from the MPI Fortran code that is available in the "NPB3.3-MPI" suite and is not heavily optimized for the GPU architecture. For example, Figure 6.14 shows the relative execution time of the single device version of the benchmarks on the CPU and the GPU in our experimental system. We can see that most of the benchmarks run better on the CPU but the degree of speedup varies, whereas EP runs faster on the GPU. This means that the device with the highest theoretical peak performance and bandwidth, that is the GPU, is not always the best choice for the given

kernel.

Each SNU-NPB-MD benchmark has specific restrictions on the number of command queues that can be used depending on its data and task decomposition strategies, as documented in Table 6.4. Also, the amount of work assigned per command queue differs per benchmark, that is, some create constant work per application and work per command queue decreases for more queues while others create constant work per command queue and so, work per application increases for more queues. In order to use more command queues than the available devices in the program, one could write a simple round robin queue-device scheduler, but an in-depth understanding of the device architecture and node topology is needed for ideal scheduling. Also, some kernels have different device-specific launch configuration requirements depending on the resource limits of the target devices, and by default, these configurations are specified only at kernel launch time. Moreover, such kernels are conditionally launched with different configurations depending on the device type (CPU or GPU). In order to *dynamically* choose the ideal kernel-device mapping, a scheduler will need the launch configuration information for all the target devices before the actual launch itself.

We enable MultiCL's dynamic command-queue scheduling by making the following simple code extensions to each benchmark: (1) we set the desired scheduling policy to the context during context creation, and (2) we set individual command-queue properties as runtime hints at command-queue creation or around explicit code regions. In some kernels, we also use the `clSetKernelWorkGroupInfo` function to separately express the device-specific kernel launch configurations to the runtime, so that the scheduler can have the flexibility to model the kernel for a particular device along with the correct corresponding kernel launch configuration. These simple code changes, together with the MultiCL runtime optimizations, enable the benchmarks to be executed with ideal queue-device mapping. The application developer has to only think about the data-task decompositions among the chosen number of command queues, while at the same time, not worry about the underlying node architecture.

Table 6.4 also shows our chosen MultiCL scheduler options for the different benchmarks. The iterative benchmarks typically have a 'warmup' phase during the loop iterations, and we consider them to be ideal candidates for *explicit* kernel profiling because they form the most representative set of commands that will be consistently submitted to the target command queues. For such iterative benchmarks, we set the command queues with the

Table 6.4: Summary of SNU-NPB-MD benchmarks, their requirements and our custom scheduler options.

Bench.	Classes	Cmd. Queues	MultiCL Scheduler Option(s)
BT	S,W,A,B	Square: 1,4	SCHED_EXPLICIT_REGION, clSetKernelWorkGroupInfo
CG	S,W,A,B,C	Power of 2: 1,2,4	SCHED_EXPLICIT_REGION
EP	S,W,A,B,C,D	Any: 1,2,4	SCHED_KERNEL_EPOCH, SCHED_COMPUTE_BOUND
FT	S,W,A	Power of 2: 1,2,4	SCHED_EXPLICIT_REGION, clSetKernelWorkGroupInfo
MG	S,W,A,B	Power of 2: 1,2,4	SCHED_EXPLICIT_REGION
SP	S,W,A,B,C	Square: 1,4	SCHED_EXPLICIT_REGION

SCHED_EXPLICIT_REGION property at creation time and trigger the scheduler explicitly around the warmup code region. We call clSetCommandQueueProperty with SCHED_AUTO and SCHED_OFF flags to start and stop scheduling respectively. Other code regions were not considered for explicit profiling and scheduling, because they did not form the most representative command set of the benchmark. We also did not choose the SCHED_KERNEL_EPOCH option for iterative benchmarks, because the warmup region spanned across multiple kernel epochs and the aggregate profile of the region helped to generate the ideal queue-device mapping. On the other hand, the EP benchmark (random number generator) is known to be very compute-intensive and not iterative. At command-queue creation time, we simply set the SCHED_KERNEL_EPOCH and SCHED_COMPUTE_INTENSIVE properties as runtime hints, which are valid for the queue's lifetime. In the BT and FT benchmarks, we additionally use our proposed clSetKernelWorkGroupInfo OpenCL API (Section 4.2) to set CPU- and GPU-specific kernel launch parameters. The parameters that are later passed to clEnqueueNDRangeKernel are ignored by the runtime. This approach decouples the kernel launch from a particular device, thus enabling the runtime to dynamically launch kernels on the ideal device with the right device-specific kernel launch configuration.

We evaluate each benchmark with problem sizes from the smallest (S) to the largest problem size that fits on each available device. Figure 6.15 shows a performance comparison of MultiCL-based automatic scheduling with manual round-robin techniques as the baseline. The benchmark class in the figure denotes the largest problem size for that application that could fit on the device memories, and each benchmark uses four command queues. One can schedule four queues among three devices (2 GPUs and 1 CPU) in 3^4 ways, but for our demonstration purpose, we showcase a few explicit schedules that we consider are more likely to be explored by users – (1) CPU-only assigns all four command queues to the CPU, (2) GPU-only assigns all four command queues to one of the GPUs, (3) Round-robin

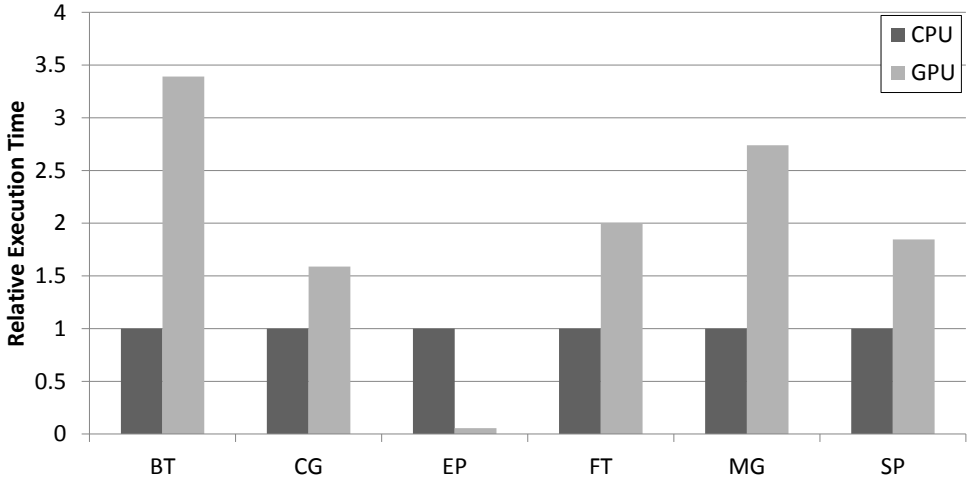


Figure 6.14: Relative execution times of the SNU-NPB benchmarks on CPU vs. GPU.

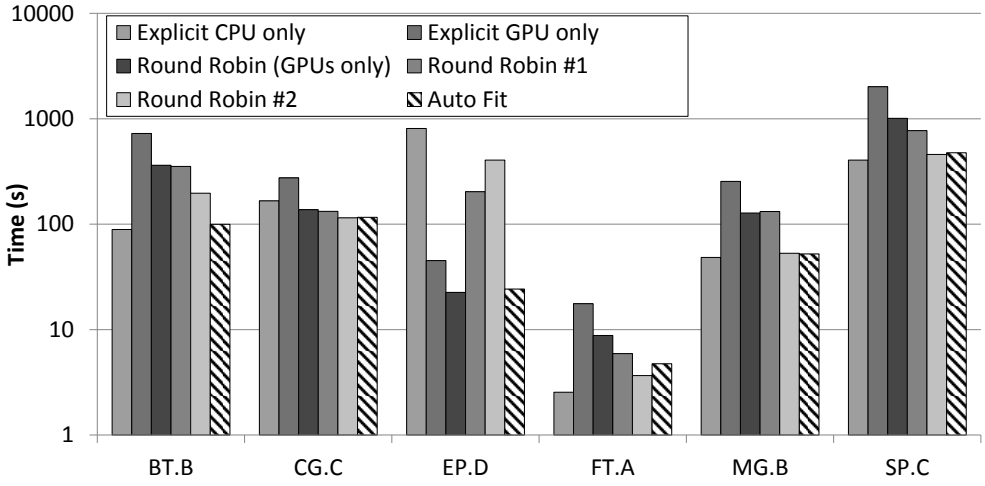


Figure 6.15: Performance overview of SNU-NPB (MD) for manual and MultiCL's automatic scheduling. Number of command queues: 4. Available devices: 1 CPU and 2 GPUs.

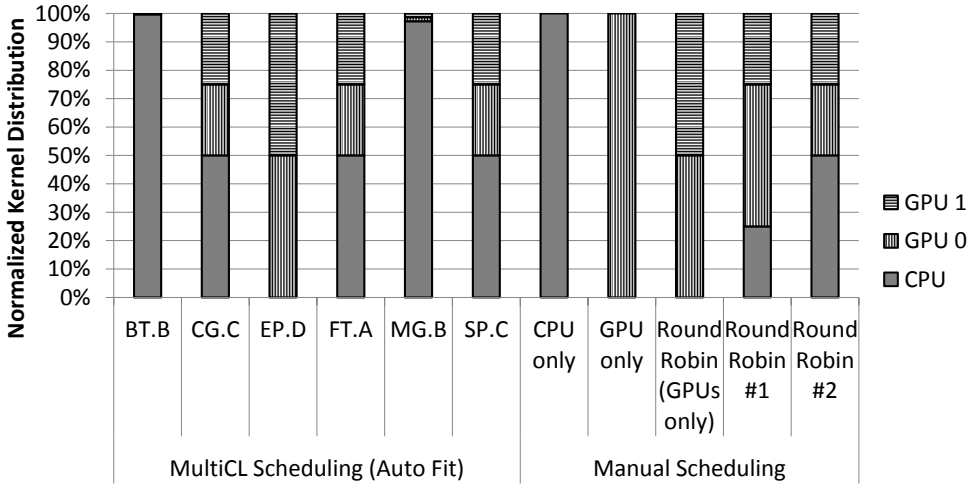


Figure 6.16: Distribution of SNU-NPB (MD) kernels to devices for manual and MultiCL's automatic scheduling. Number of command queues: 4. Available devices: 1 CPU and 2 GPUs.

(GPUs) assigns two queues each to the two GPUs, (4) Round-robin #1 assigns two queues to one GPU, one queue to the other GPU and one queue to the CPU, and (5) Round-robin #2 assigns two queues to the CPU and one queue to each GPU. Given that five benchmarks perform better on the CPU and EP works best on the GPU, we consider some of the above five schedules to form the best and worst queue-device mappings and expect the MultiCL scheduler to automatically find the best queue-device mapping.

We define the modeling overhead of our scheduler as the difference between the performance obtained from the ideal queue-device mapping and the performance obtained from the scheduler driven queue-device mapping, expressed as a percentage of the ideal performance, that is $\frac{T_{scheduler\_map} - T_{ideal\_map}}{T_{ideal\_map}} * 100$. Figure 6.15 shows that automatic scheduling using the MultiCL runtime achieves near optimal performances, which indirectly means ideal queue-device mapping, and the geometric mean of the overall performance overhead is 10.1%. The overhead of FT is more than the other benchmarks, which we analyze in the next paragraph. Figure 6.16 shows how the performance model in MultiCL's scheduler has distributed the kernels among the available devices. A close comparison with the benchmarks' CPU vs. GPU performance from Figure 6.14 indicates that our scheduler maps queues to devices in a near ideal manner. For example, Figure 6.14 indicates that the BT and MG benchmarks perform much better on the CPU than the GPU, and Figure 6.16 indicates that our scheduler has assigned most of the kernels from all iterations to the CPU

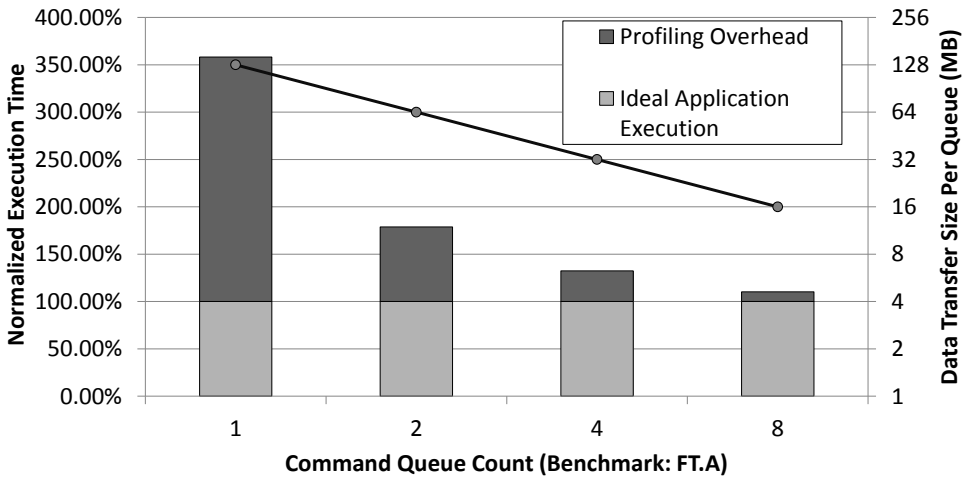


Figure 6.17: Data transfer overhead for the FT (Class A) benchmark.

and almost none to the GPU. Similarly, EP performs best on the GPU (Figure 6.14) and we see that our scheduler has assigned all kernels to the GPU. The other benchmarks are still better on the CPU, but to a lesser degree, and thus we see that the CPU still gets a majority of the kernels, but the GPUs too get their share of work. We see similar trends for the other problem classes and other command queue numbers as well.

Effect of Data Transfer Overhead in Scheduling

The FT benchmark distributes the input data among the available command queues, that is, data per queue decreases as the number of queues increase. MultiCL's kernel profiling module copies the input data only once per device for performance modeling; hence, the cost is amortized for more command queues, and our modeling overhead reduces. While Figure 6.15 indicates that the modeling overhead in FT is about 45% when compared to the ideal queue-device mapping and when four command queues are used, Figure 6.17 indicates that the modeling overhead decreases with increasing command queues. While the other benchmarks work on similar data footprints in memory, they do not transfer as much data as FT, and thus exhibit apparently negligible data transfer overhead while scheduling.

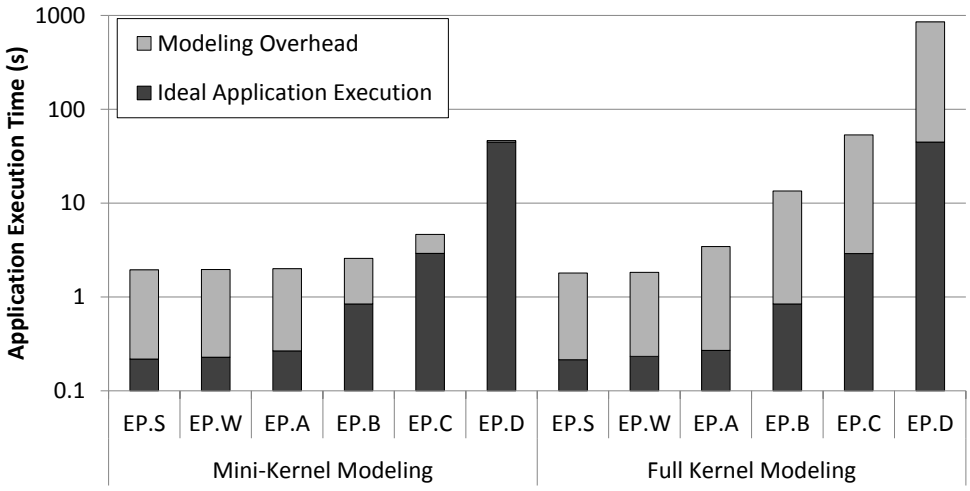


Figure 6.18: Impact of mini-kernel modeling for the EP benchmark.

Effect of Mini-kernel Profiling in Scheduling

The EP benchmark does random number generation on each device and is very compute-intensive, and the CPU (non-ideal device) can be up to $20\times$ slower than the GPU (ideal device) for certain problem sizes. Since the full kernel-modeling approach runs the entire kernel on each device before device selection, the runtime overhead when compared to the ideal device combination can also be about $20\times$. Moreover, running the full kernel means that the modeling overhead increases for larger problem sizes. On the other hand, our mini-kernel modeling approach just runs a single workgroup on each device, which incurs a *constant* modeling overhead for any problem size. Mini-kernel modeling thus dramatically reduces the modeling overhead to only about 3% of the total execution time for large problem sizes, while making optimal device mapping. We perform mini-kernel profiling for all the other benchmarks as well, but since they are not as compute-intensive as EP, its apparent benefits are negligible.

Summary

With MultiCL, we make parametric changes to at most four OpenCL functions in existing benchmarks and trigger the MultiCL scheduler to automatically schedule the command queues and map them to the ideal combination of devices. We choose the auto fit global scheduler for the context, while the command queues choose either the explicit region

or kernel epoch local scheduler options. The MultiCL scheduler performs static device profiling to collect the device distance metrics, then performs dynamic kernel profiling to estimate the kernel running costs, and then computes the aggregate cost metric from the data transfer and kernel execution costs. We derive the data transfer costs based on the device profiles and the kernel profiles provide the kernel execution costs. We use the aggregate cost metric to compute the ideal queue-device mapping. Optimal command-queue-to-device mapping is automatically accomplished with a reasonably low performance overhead and minor code modifications.

6.5.2 Seismology Modeling Simulation

FDM-Seismology is 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 [53]. The application implements a parallel velocity-stress, staggered-grid finite-difference method 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 is used for each grid cell. Since the computational domain is truncated in order to keep the computation tractable, absorbing boundary conditions are placed around the region of interest to keep the reflections minimal when boundaries are impinged by the outgoing waves. This strategy helps simulate unbounded domains. The simulation iteratively computes the velocity and stress wavefields within a given subdomain. Moreover, the wavefields are divided into two independent regions, and each region can be computed in parallel. The reference code of this simulation is written in Fortran [61].

Design and Implementation in OpenCL

For our experiments, we extend an existing OpenCL implementation [43] of the FDM-Seismology simulation as the baseline. The OpenCL implementation divides the kernels into velocity kernels and stress kernels, where each set of kernels computes the respective wavefields at its two regions. The velocity wavefields are computed by using seven OpenCL kernels, three of which are used to compute on region-1 and the other four kernels to compute on region-2. Similarly, the stress wavefields are computed by using 25 OpenCL kernels, 11 of which compute on region-1 and 14 kernels compute on region-2. We have two OpenCL implementations of the simulation: (1) *column-major data*,

which directly follows Fortran's column major array structures, and (2) *row-major data*, which uses row major array structures and is more amenable for GPU execution. Moreover, since the two wavefield regions can be computed independently, their corresponding kernels are enqueued to separate command queues.

In our experimental system, the two command queues can be scheduled on the three OpenCL devices in 3^2 different ways. Figure 6.19 demonstrates the performance of both versions of the kernels on different device combinations. We see that the column-major version performs best when all the kernels are run on a single CPU and performs worst when all of them are run on a single GPU, and the performance difference between the two queue-device mappings is $2.7\times$. On the other hand, the row-major version is best when the command queues are distributed across two GPUs and is $2.3\times$ better than the performance from the worst-case mapping of all kernels on a single CPU.

We compare the performance of MultiCL's two global schedulers, round robin and auto fit, by simply setting the context property to either the `ROUND_ROBIN` or `AUTO_FIT` respectively. FDM-Seismology has regular computation per iteration, and each iteration consists of a single synchronization epoch of kernels. So, as our local scheduler, we can either choose the implicit `SCHED_KERNEL_EPOCH` at queue creation time, or choose the `SCHED_EXPLICIT_REGION` and turn on automatic scheduling explicitly just for the first iteration by using `clSetCommandQueueProperty`. We use the `SCHED_KERNEL_EPOCH` option in our experiments, but the final mapping and modeling overhead is expected to be the same for the other option as well. Figure 6.19 shows that MultiCL's auto fit scheduler maps the devices optimally for both code versions. The performance of the auto fit case is similar to the CPU-only case for the column-major code and is similar to the dual-GPU case for the row-major version of the code, with a negligible modeling overhead of less than 0.5%. On the other hand, the round-robin scheduler always chooses to split the kernels among the two GPUs, which does not provide us the best combination for the column-order version of the code. Figure 6.20 shows that for the auto-fit scheduler, while the first iteration incurs runtime overhead, the added cost gets amortized over the remaining iterations.

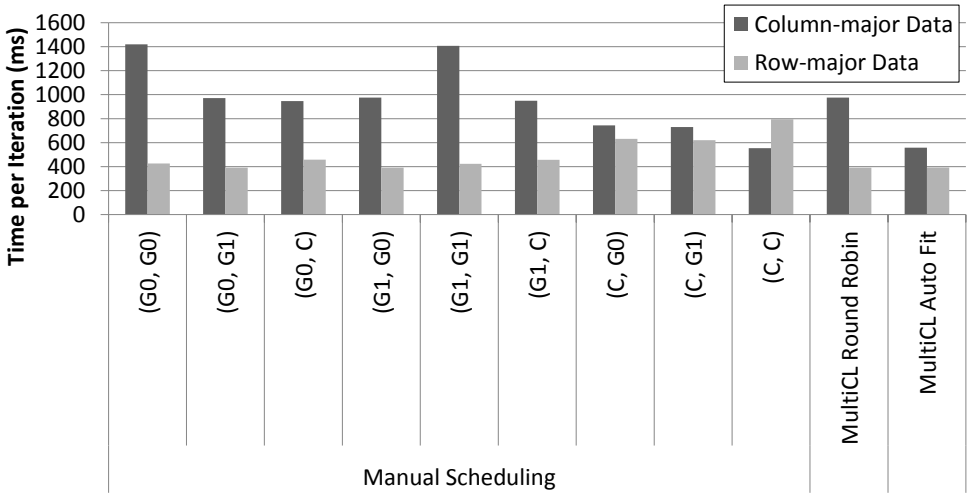


Figure 6.19: FDM-Seismology performance overview.

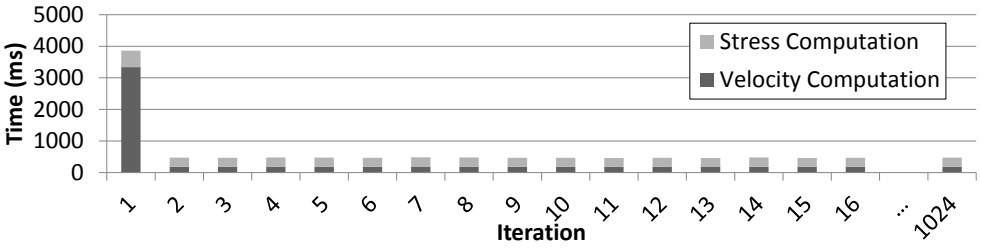


Figure 6.20: FDM-Seismology performance details for the `AUTO_FIT` scheduler. The graph shows that the overhead of performance modeling decreases asymptotically with more iterations.

Data Marshaling with OpenCL

The FDM-Seismology simulation has data marshaling phases after both the velocity and stress compute phases. The default version of the code performed data marshaling on the CPU. In this section, we describe the consequences of performing the data marshaling operations on one of the OpenCL devices, and how the MultiCL scheduler adapts to the changed workflow.

We have reimplemented all the GPU data marshaling kernels from CUDA (Section 5.3.2) to OpenCL for this purpose. In the CPU-based data marshaling scenario, the region-1 and region-2 kernels were computed concurrently using two separate command queues. However, the data marshaling kernels are consecutive and work on both regions simultaneously, i.e. there is data dependency between regions 1 and 2. To address this data dependency, we create a third separate command queue that is used to invoke all the data marshaling kernels. We discuss the row-major data execution mode in this section, but the column-major scenario works similarly.

As a consequence to performing data marshaling on the device, the host-device bulk data transfers before and after each velocity-stress computation kernel are completely avoided. The need to explicitly bulk transfer data from the device to the host arises only at the end of the iteration, when the results are transferred to the host to be written to a file (Figure 6.21). Furthermore, we can use MPI-ACC to perform these data transfers directly among OpenCL devices, which we discuss in section 6.6. In summary, the application uses three command queues and the MultiCL runtime handles scheduling them to the available devices.

After every computation phase, the marshaling queue synchronizes with the computation queues, which involves both execution and data synchronizations. In the CPU-based data marshaling scenario, the two computation queues are independently scheduled on different OpenCL devices (GPUs) for the entire duration of the application and there was no cross-queue synchronization. However, the device-based data marshaling scenario required explicit cross-queue synchronization before and after marshaling. The SnuCL runtime automatically migrates the required data to and from the target device around the marshaling phase. While splitting the velocity and stress kernels across two devices reduces the computation cost, the marshaling kernels execute on a single device and incur *data resolution* cost, which is

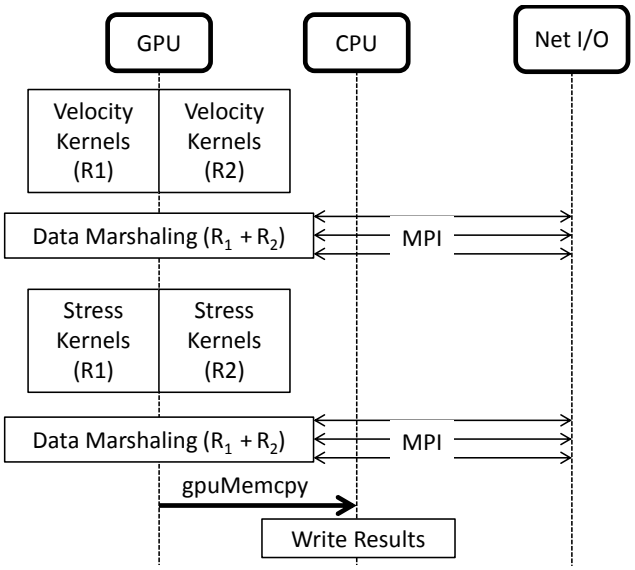


Figure 6.21: Data marshaling with OpenCL in FDM-Seismology. Data dependency: stress and velocity kernels work concurrently with independent regions on separate command queues, whereas the data marshaling step works with both regions on a single command queue.

required to maintain data consistency across the different devices. So, the question is if it is worthwhile to schedule the compute command queues across two devices and incur data resolution cost, or is it better to schedule all the queues to a single device and completely avoid the data resolution cost? Our MultiCL runtime schedules all the available command queues so that the *overall execution time*, which includes the kernel execution cost and the data resolution cost for marshaling, is minimized.

Evaluation

In this section, we evaluate the tradeoff between scheduling on multiple devices to reduce the kernel execution (compute) time vs. scheduling on a single device to reduce the data resolution time. To this end, we implement a seismology "mini-application", which captures the core computation-communication pattern of FDM-Seismology. Figure 6.22 describes the pseudo-code of the mini-application. It begins with a compute phase that consists of a compute-intensive kernel performing a series of multiple and multiply-add operations on two separate data sets on two independent command queues. The compute phase is followed by a data marshaling phase, which consists of a lightweight kernel

```
1  // Initialize command queues
2  cl_command_queue queue1, queue2, queue3;
3
4  // Mini-application loop
5  for (...) {
6    // Two independent compute loops
7    compute_data_1(data1, queue1);
8    compute_data_2(data2, queue2);
9    sync(queue1);
10   sync(queue2);
11   // Data resolution happens before marshaling
12   marshal_data(data1, data2, queue3);
13 }
```

Figure 6.22: Pseudo-code of the seismology mini-application.

running on a single command queue and takes negligible cycles to compute. However, the marshaling kernel references both the computed data sets to induce data dependency among the command queues. In summary, the compute phase is compute-intensive and works independently on two data sets, wheres the marshaling phase is not compute-intensive but has to resolve dependencies and incurs a data resolution cost. The compute and marshaling phases are executed for multiple iterations and the average running time is reported. We can control the relative computation and data resolution costs by modifying the computation loops and working data set sizes respectively.

MultiCL's `AUTO_FIT` scheduler optimizes for minimal total execution time. If the concurrent kernel execution time on two devices (T_K2), in addition to the data resolution time (T_DR) is lesser than the consecutive kernel execution time on a single device (T_K1), then the scheduler will map the compute command queues to separate devices. If the consecutive kernel execution time on a single device (T_K1) is lesser, then the scheduler maps all the command queues to a single device.

In our experiment, we keep the compute phase constant and only vary the working data set size, so that the data resolution cost increases for larger working data sets, whereas the compute phase time remains constant for all working data sets. Figure 6.23 shows the time decomposition for kernel execution and data resolution costs for all queue-GPU combinations. We ignore the CPU device in this case because the GPUs are about $100\times$ faster than the CPU for this specific mini-application example, and are irrelevant in our scheduler analysis. We see from a detailed analysis in figure 6.24 that for smaller data sizes, $T\_K1 > T\_K2 + T\_DR$ and two devices are better to schedule the queues despite the data resolution cost. For larger data sizes, $T\_K1 < T\_K2 + T\_DR$ and the data resolution cost of using

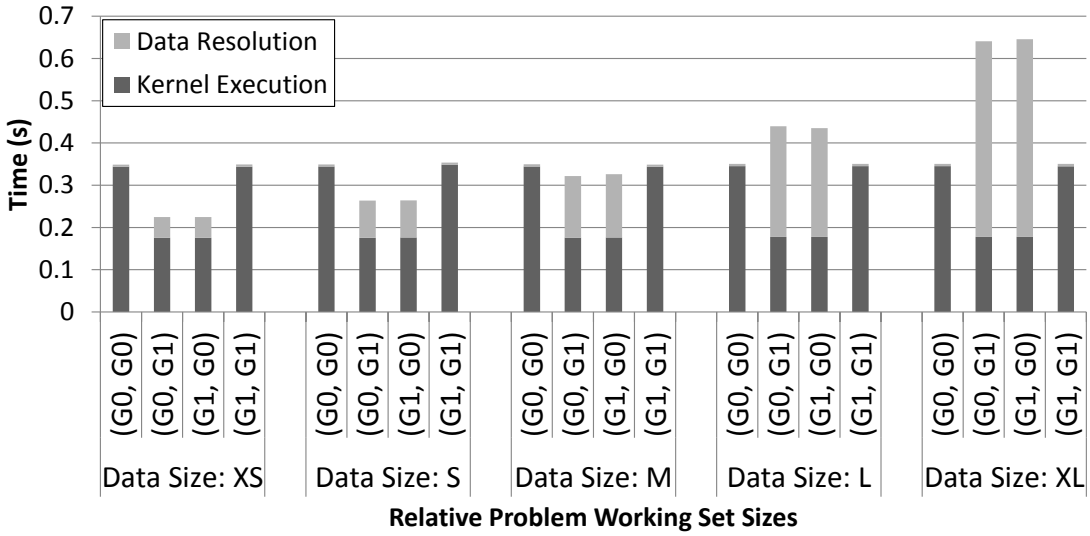


Figure 6.23: Seismology mini-application performance overview on all queue-GPU combinations. Two GPUs are better for smaller data sizes, whereas single GPU is better for larger data sizes.

two devices outweigh the advantage of concurrent processing, and running everything on a single device performs better overall. As the data sizes grow, the case for a single device schedule becomes stronger as well.

While our mini-application was controlled to have a constant compute phase for all working data sets, the compute phase may vary in real applications. We see from a detailed analysis in figure 6.26 that for all data sizes, $T\_K1 < T\_K2 + T\_DR$ and the data resolution cost of using two devices outweigh the advantage of concurrent processing, and running everything on a single device performs better overall. This is because the compute phase time also varies with the working data set size and there is no tradeoff point in the positive X-axis where two devices would be better than a single device. As the data sizes grow, the case for a single device schedule becomes stronger as well.

Figure 6.25 shows the time decomposition for kernel execution and data resolution costs for all queue-GPU combinations. We note that while the case for a single GPU may become better for smaller data sizes, the application has a non-negligible sequential code, which limits the overall parallelism of the application and using multiple GPUs make lesser sense. Figure 6.25 also shows that our MultiCL scheduler correctly identifies the above trends in the device mapper module of the runtime, and assigns all command queues to a single device for all data sets.

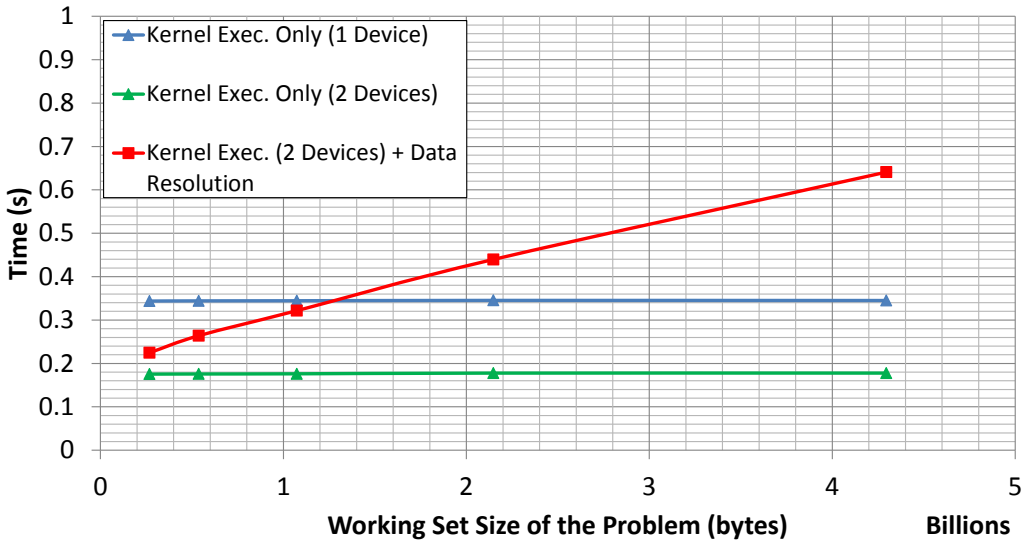


Figure 6.24: Seismology mini-application performance analysis for single device vs. two device configurations. For smaller data sizes, $T\_K1 > T\_K2 + T\_DR$, i.e. two devices are better. For larger data sizes, $T\_K1 < T\_K2 + T\_DR$, i.e. a single device is better.

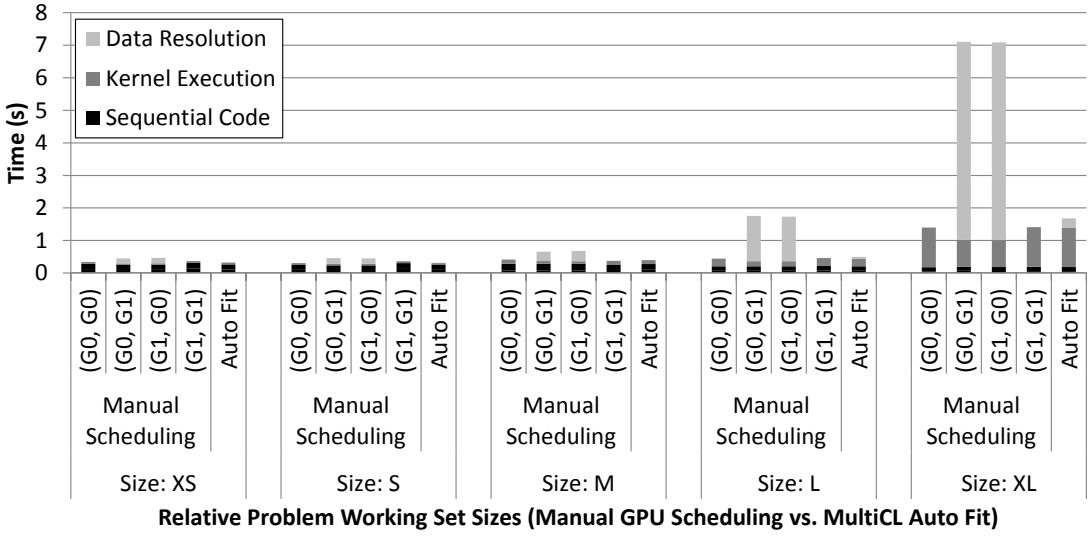


Figure 6.25: FDM-Seismology application performance overview on all queue-GPU combinations. For all data sizes, a single GPU is better than using two GPUs. The sequential code cost also provides lesser incentive to move to multiple GPUs.

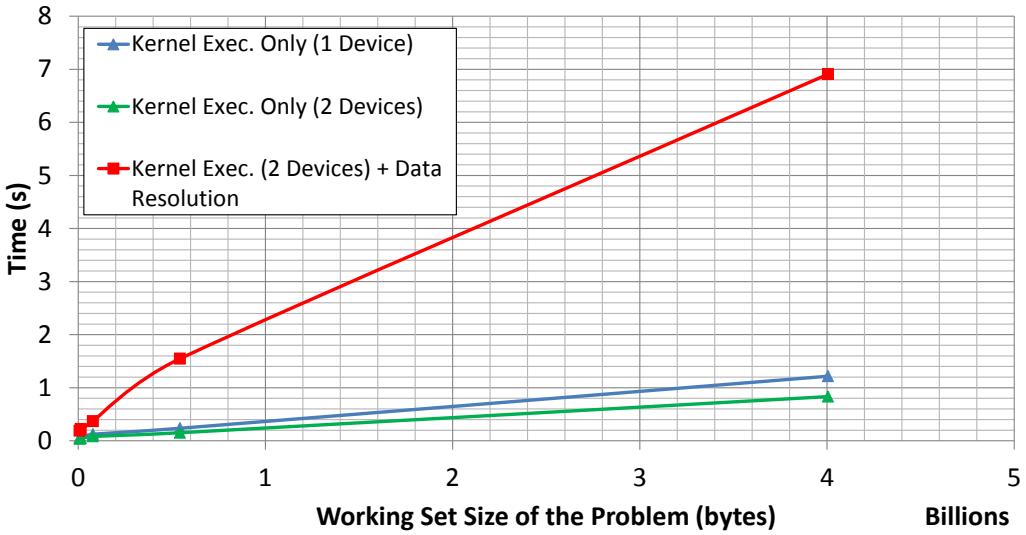


Figure 6.26: FDM-Seismology application performance analysis for single device vs. two device configurations. For all data sizes, $T\_K1 < T\_K2 + T\_DR$, i.e. a single device is better.

6.5.3 Programmability Benefits

For all our experiments, we made minimal code changes to the application code, with an average modification of about *four* lines of code to the entire program.[3] The user is required to add new context properties to set the global scheduling policy and set the command queue properties for local policies and runtime scheduler hints. The other runtime features are optional, such as using `clSetCommandQueueProperty` to explicitly control the local policy and `clSetKernelWorkGroupInfo` to set device-specific kernel launch configurations. We have shown that with minimal code changes to a given OpenCL program, our scheduler can automatically map the command queues to the optimal set of devices thereby significantly enhancing the programmability for a wide range of benchmarks and real-world applications. Our scheduler is shown to incur negligible overhead for the seismology simulation.

[3]We assume that each OpenCL function call can fit in one source line.

6.6 Discussion: Codesigning the Data Movement and Task Scheduler Runtimes

The data movement runtime orchestrates efficient data communication among CPUs and accelerators across nodes. On the other hand, the task management runtime schedules tasks within a node. Ideally, the data communication runtime should be independent of the within-node task management runtime, where data pieces that are required for pipelining or direct communication are fetched through the device-specific programming model, in our case, the command queue or the GPU stream. The data movement runtime should not rely on device specific data structures, and instead use device abstractions like command queues to do data communication. However, the current GPU drivers limit the host side pinned memory pool to be shared among all the available devices, which means that we have to create device specific pinned memory in the MPI-ACC runtime. While MPI-ACC issues intermediate data transfer commands to the command queue and not directly to the device, it still needs to query the actual device that is associated with the command queue to decide the specific memory pool that should be used for pipelining. If all the vendor drivers were able to share a common pinned memory pool for RDMA transfers, then the MPI-ACC runtime could have simply chosen any generic host buffer for pipelining and continue to interface with just the high level command queue device abstractions. Querying for the dynamic command queue's current device is not supported by the OpenCL standard, and so we extend the OpenCL specification to include a "latest" device query flag to retrieve the device with the latest copy of the memory object of interest.

We use point-to-point latency benchmarks and the FDM-Seismology application to evaluate our codesign of the within- and across-node layers of MPI-ACC. We implement FDM-Seismology application in OpenCL, which includes all the velocity and stress computations as well as the marshaling kernels on the GPU device. Using MPI-ACC, we can perform end-to-end device data transfers, while the MultiCL runtime is used to dynamically choose the ideal device for the set of velocity and stress kernels.

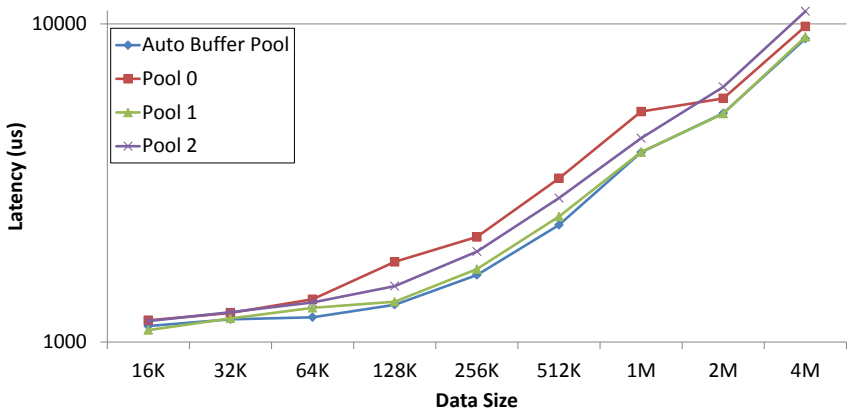


Figure 6.27: Point-to-point communication between OpenCL Device-1 on each node. Best performance is achieved if the intermediate buffers for pipelining also correspond to the same device. MPI-ACC automatically chooses the right buffer pool by querying the queue's latest device.

Send-Receive Microbenchmark We measure the performance impact of buffer pool choices when moving data between two remote OpenCL devices using MPI-ACC. In this experiment, we arbitrarily choose device 1 on each node and perform a ping-pong latency test by modifying the OSU latency benchmark. Figure 6.27 shows that our codesign of MPI and the OpenCL runtimes help in choosing the right set of buffer pool to perform pipelining, and thus better performance.

FDM-Seismology with MPI and OpenCL In this section, we evaluate the multi-node configuration of FDM-Seismology using both the MPI-ACC and MultiCL runtimes. Our MPI-based parallel version of the application divides the input finite difference model into submodels along different axes such that each submodel can be computed on different CPUs (or nodes). This domain decomposition technique helps 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. Each processor updates both regions of its own wavefields after receiving all its neighbors' wavefields.

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 post-communication 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 finite difference (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. Data marshaling can be performed either on the host or on the device, as discussed in the previous section.

With MPI-ACC as the communication library, we still perform data marshaling on the GPU, but communicate the marshaled data directly to and from the GPU without explicitly using the CPU for data staging. Also, the bulk transfer of data still happens only once at the end of each iteration to write the results to a file. But, the data marshaling step happens multiple times during a single iteration and consequently, the application launches a series of GPU kernels. While consecutive kernels entail launch and synchronization overhead per kernel invocation, the benefits of faster data marshaling on the GPU and optimized MPI communication outweigh the kernel overheads.

We extend the single node implementation to include MPI-ACC calls to exchange the marshaled data with the neighbors. Once the data is marshaled on the OpenCL devices, the send-receive calls between the nodes are associated with the marshaling command queue (Figure 6.28).

While the MultiCL scheduler determines the local mapping of command queues with devices, the MPI-ACC runtime simply interfaces with the high level OpenCL command queues and performs data pipelining across the network. As previously mentioned, MPI-ACC only queries for the scheduled device to determine the appropriate buffer pool for pipelining.

```
1  // Initialize all command queues
2  cl_command_queue queue1, queue2, queue3;
3  // Initialize event object
4  cl_event event;
5
6  // OpenCL extensions for automatic device selection
7  queue1 = clCreateCommandQueue(context, dev, CL_QUEUE_AUTO_DEVICE, ...);
8  queue2 = clCreateCommandQueue(context, dev, CL_QUEUE_AUTO_DEVICE, ...);
9  queue3 = clCreateCommandQueue(context, dev, CL_QUEUE_AUTO_DEVICE, ...);
10
11 // Initialize MPI data types for MPI-ACC
12 MPI_Type_dup(MPI_CHAR, &new_type);
13 MPI_Type_set_attr(new_type, BUF_TYPE, BUF_TYPE_OCL);
14 // Main application loop
15 for (...) {
16   // Velocity Computations
17   compute_velocity_region_1(queue1);
18   compute_velocity_region_2(queue2);
19   sync(queue1);
20   sync(queue2);
21   marshal_velocity_data(queue3, event);
22   // Automatic end-to-end data transfer and synchronization with MPI-ACC
23   MPI_Type_set_attr(new_type, COMMAND_QUEUE, queue3);
24   MPI_Type_set_attr(new_type, EVENT, event);
25   // MPI-ACC call
26   MPI_Isend(buf, new_type, neighbor, ...);
27   MPI_Irecv(buf, new_type, neighbor, ...);
28   MPI_Waitall(...);
29   // Stress Computations
30   compute_stress_region_1(queue1);
31   compute_stress_region_2(queue2);
32   sync(queue1);
33   sync(queue2);
34   marshal_stress_data(queue3);
35   // Automatic end-to-end data transfer and synchronization with MPI-ACC
36   MPI_Type_set_attr(new_type, COMMAND_QUEUE, queue3);
37   MPI_Type_set_attr(new_type, EVENT, event);
38   //MPI-ACC
39   MPI_Isend(buf, new_type, neighbor, ...);
40   MPI_Irecv(buf, new_type, neighbor, ...);
41   MPI_Waitall(...);
42 }
43 MPI_Type_free(&new_type);
```

Figure 6.28: Pseudo-code of FDM-Seismology using MPI-ACC and MultiCL runtimes.

6.7 Conclusion

We present our concluding thoughts on performance modeling and projection, and also our experiences in designing and implementing a runtime system for task-device mapping.

6.7.1 Performance Modeling and Projection

We proposed, implemented, and evaluated an online performance projection framework for optimal 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 offline device profiling and online kernel characterization. To automatically obtain runtime kernel statistics with an asymptotically lower overhead, we propose the mini-emulation technique that functionally simulates a single workgroup to collect per-workgroup statistics, which can then be used to calculate full-kernel statistics. Our technique is especially suitable for online performance projection for kernels with large data sets. Our experiments with GPU devices of different vendors show our technique is able to select the optimal device in most cases.

6.7.2 Task-Mapping Runtime

We introduced MultiCL, our task-mapping runtime, which includes automatic command-queue scheduling capabilities. Our proposed runtime flags can be used to control the scheduling both globally at the context level and locally at the command queue level. Our runtime optimizations enable users to focus on application-level data and task decomposition and not worry about device-level architectural details and scheduling. Our runtime scheduler includes static device profiling, dynamic kernel profiling, and online device mapping. We design novel overhead reduction strategies including mini-kernel modeling for compute-intensive kernels, reduced data transfers, and profile data caching for future kernel invocations. Our experiments on the NPB benchmarks and a real-world seismology simulation (FDM-Seismology) demonstrate that the MultiCL runtime scheduler always maps command queues to the optimal device combination and has an average runtime overhead of 10% for the NPB benchmarks and negligible overhead for

FDM-Seismology. We also discuss and evaluate our codesigning efforts of integrating the across node communication runtime with the within node task scheduler.

Chapter 7

Conclusions

This dissertation pushed the boundaries of the MPI+X programming model (where X = CUDA, OpenCL, OpenMP, OpenACC, etc.) and its associated runtime systems for high-performance clusters with heterogeneous computing devices. We conclude our document with some related research directions and concluding thoughts.

7.1 Related Research Directions

In this section, we describe our work's applicability in relation to higher level within-node "X" programming models, and note few directions in which this research can be pursued in the future.

7.1.1 Interoperability With Other "X" Programming Models

Our dissertation focuses on MPI+X programming models, where X was device offload models, like CUDA and OpenCL. Here, we discuss the interoperability of directive-based programming models, such as OpenMP [7] and OpenACC [8], with MPI, and how MPI-ACC may be used to enhance the interoperability.

OpenMP (v4.0) [7] and OpenACC [8] are directive-based programming models for writing parallel programs primarily

for CPUs and GPUs in single node systems. The programmer writes a serial program with annotations to denote parallelism, heterogeneity and asynchrony. A compiler translates the annotated code to run on the multiple CPU cores or one of the available GPU devices. All the above are fork-join models where a single master thread executes sequentially until a parallel region construct is encountered. The parallel regions fork worker threads on the CPU or GPU kernels or both, then join back to the master thread to continue the serial execution. Data management in the annotated regions are *host-centric*, i.e. data is usually copied in from the host to the device at the beginning of the region and copied out at the end of the region to maintain data consistency. Special clauses exist in both OpenMP and OpenACC to perform just copying in or copying out, which provides better data locality and consistency control to the programmer.

MPI-ACC extends MPI by allowing direct end-to-end data communication among CPUs and other devices. MPI-ACC also performs automatic synchronization between device execution and data movement. For full interoperability of MPI-ACC with directive-based models, the data access and synchronization semantics of device data and device tasks must to be clearly defined by the OpenMP's and OpenACC's specifications.

Using MPI-ACC outside the region The OpenMP/OpenACC models are primarily host-centric, which works well when used in conjunction with vanilla MPI, because MPI functions also operate only on host memory. If OpenMP and OpenACC are also device-centric and directly operate on device buffers, then the programmer can leverage MPI-ACC and its advanced runtime optimizations for data movement. The resulting will look very similar to a CPU-only scenario, where MPI-ACC transfers data to the target device, and an annotated loop is executed on the destination device by assuming that the incoming data is already available on the device and is valid. OpenACC's `deviceptr()` clause is a mechanism to tell the compiler to treat the incoming pointer as a device pointer and not a host pointer, and can be used with MPI-ACC on CUDA buffers. Figure 7.1 shows a sample pseudo-code, where MPI-ACC is used to receive CUDA data into a node, and an OpenACC loop operates on the received data. However, the `deviceptr` clause requires the device data to be represented by `void *`, and so this approach will not work with OpenCL buffers and MPI-ACC.

```
1  char *d_ptr;
2  cudaMalloc((void *)&d_ptr, sz_bytes);
3  MPI_Type_dup(MPI_CHAR, &type);
4  MPI_Type_set_attr(type, BUF_TYPE, BUF_TYPE_CUDA);
5  MPI_Recv(d_ptr, type, ...);
6  MPI_Type_free(&type);
7  /* MPI-ACC is used to directly receive d_ptr to the device */
8  #pragma acc data deviceptr(d_ptr)
9   #pragma acc parallel
10   /* d_ptr is directly operated upon in the region
11      without any explicit host-device copies */
12   foo(d_ptr);
13 /* Rest of the program */
```

Figure 7.1: Interoperability of OpenACC with MPI-ACC: device data access within a region.

```
1  char *d_ptr = malloc(sz_bytes);
2  #pragma acc data copy(d_ptr)
3  {
4      // MPI-ACC, a host method, is invoked in this region
5      // by directly operating on the device data pointer
6      #pragma acc host_data use_device(d_ptr)
7      {
8          MPI_Type_dup(MPI_CHAR, &type);
9          MPI_Type_set_attr(type, BUF_TYPE, BUF_TYPE_CUDA);
10         MPI_Send(d_ptr, type, ...);
11         MPI_Type_free(&type);
12     }
13     /* Other OpenACC code */
14 }
15 /* Rest of the program */
```

Figure 7.2: Interoperability of OpenACC with MPI-ACC: MPI-ACC from within a region.

The latest OpenMP v4.0 specification does not address the scenario of using device pointers directly within a loop region, and we want MPI-ACC and other GPU-integrated MPI solutions to motivate the OpenMP standards group to consider adding direct device access support to OpenMP.

Using MPI-ACC within a region Since MPI-ACC is implemented as a CPU library, OpenMP and OpenACC should support CPU-callable library interoperability within the annotated regions for data movement across devices. OpenACC's `acc host_data use_device()` pragma can be used to invoke MPI-ACC host calls from within a device region, and also specify that the GPU's version of the pointer be used in the MPI call. For example, figure 7.2 shows how the `acc host_data use_device()` pragma is used to send the device pointer by using MPI-ACC, which is a host method. OpenMP v4.0 does not have such a feature to enable MPI-ACC within an annotated region.

Some of the shortcomings of OpenACC's pragmas are (1) OpenCL buffers cannot be used in the MPI calls because

```
1  char *d_ptr1;
2  char *d_ptr2;
3  cudaMalloc((void *)&d_ptr1, sz_bytes);
4  cudaMalloc((void *)&d_ptr2, sz_bytes);
5  #pragma acc parallel async(1)
6  {
7      /* Operate on d_ptr1 */
8  }
9  #pragma acc parallel async(2)
10 {
11     /* Operate on d_ptr2 */
12 }
13 MPI_Type_dup(MPI_CHAR, &type);
14 MPI_Type_set_attr(type, BUF_TYPE, BUF_TYPE_CUDA);
15 /* Handling ACC_ASYNC_VAL can be implemented in MPI-ACC */
16 MPI_Type_set_attr(type, ACC_ASYNC_VAL, 1);
17 MPI_Isend(d_ptr1, type, ...);
18
19 MPI_Type_set_attr(type, ACC_ASYNC_VAL, 2);
20 MPI_Isend(d_ptr2, type, ...);
21 MPI_Type_free(&type);
22 MPI_Waitall(...);
23 /* Rest of the program */
```

Figure 7.3: Interoperability of OpenACC with MPI-ACC: synchronization.

`deviceptr()` requires that the device buffer be represented by a `void *`, and (2) the semantics of how MPI synchronizes with the gangs of worker threads that are created in the outermost region to perform data movement is still undefined. Moreover, the MPI implementation should be updated to leverage all the available threads to accelerate the MPI call itself, similar to the work done in [70].

Synchronizing MPI-ACC with OpenMP/OpenACC tasks OpenMP creates synchronous tasks by default, and there is no support yet for asynchronous GPU tasks. On the other hand, OpenACC allows asynchronous task creation and management via the `async` clause. OpenACC v2.0 supports multiple async queues, which are conceptually similar to CUDA streams, and the queues can be enumerated. If an async task has to be synchronized with an MPI-ACC call, e.g.: `MPI_Send` after the OpenACC task completes, its queue number can be passed to MPI-ACC for automatic synchronization and progress (Figure 7.3).

7.1.2 Future Work

Innovations in the MPI standard have created opportunities for better scalability of MPI applications. Of particular interest is the MPI 3.0 feature of shared-memory programming directly in MPI. All the MPI ranks in a shared-memory

node can create shared-memory windows and directly read and/or write into it with the consistency semantics of regular threading models. The scalability that can be achieved with the MPI+MPI shared memory model over the traditional MPI+OpenMP model looks very promising [37]. While this feature has been added to the MPI 3.0 standard, it has not been extended to GPU and other accelerator memories. It will be critical to explore how this programming interface would impact scalability, i.e. will the MPI+MPI model scale better than the MPI+CUDA or MPI+OpenCL models? There are also key challenges in building the runtime system, in terms of thread synchronization, efficient virtual memory management, and maintaining memory consistency semantics, while achieving acceptable performance. Current cross-vendor driver limitations may be a hindrance in making this idea realize its full potential in the short-term future.

MPI can be run natively on the Intel Xeon Phi coprocessors in one of three modes: (1) offload mode where all MPI ranks are run on the host CPU, and the application offloads the compute intensive codes on the Intel Xeon Phi corpocessor, (2) native mode where all MPI ranks are run on the Intel Xeon Phi coprocessor, and (3) symmetric mode where MPI ranks are run on both the host CPU and the Xeon Phi coprocessor. A thorough comparison of performance/programmability challenges and limitations of the above models can be compared to MPI-ACC's traditional MPI+X programming model, which includes the OpenCL-specific optimizations anyway.

AMD's Heterogeneous System Architecture (HSA) [1] provides a low-latency queueing mechanism in the user space, so that the OS is not involved in kernel launches and other data transfer commands. It will be extremely interesting to characterize the performance of MPI-ACC (with OpenCL) to be used on top of the HSA runtime, especially for real-world simulations like the FDM-Seismology application. Since the GPU access latencies will be low, small message transfers should benefit largely from our optimizations that we discussed in this document.

We can extend the MPI communication semantics to have composite locations as communication end points. For example, we can imagine a derived datatype that is defined to transfer a message that is aggregated from the main memory and all the device memories across the network. This is a very useful use-case if coscheduling across cores is performed within the node. We can investigate the state of the art in PGAS+GPU models and apply our lessons learned to those models/runtimes.

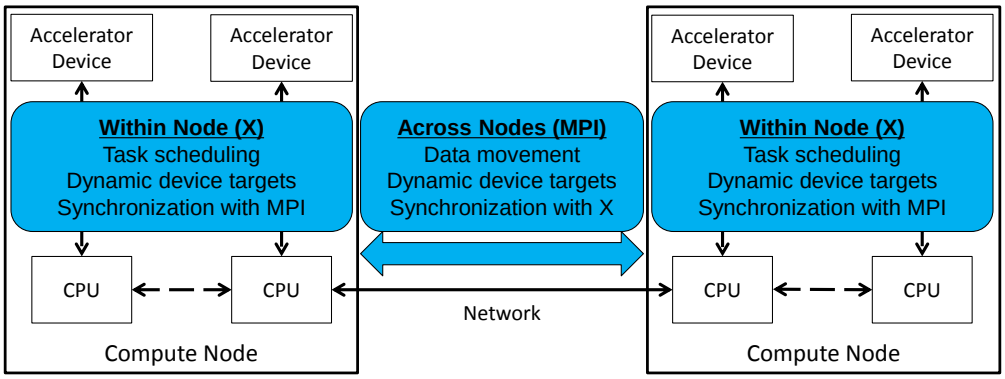


Figure 7.4: Summary of our contributions and extensions to the MPI+X programming model.

7.2 Dissertation Summary

Figure 7.4 depicts a pictorial representation of our contributions in the context of high performance clusters. The MPI+X programming model has programmability and performance challenges. By extending the MPI programming model to natively support GPU data structures and by extending the within-node GPU programming interface to enable automatic device management, we could write high-level scientific application code at scale. We encompass the inter- and intra-node runtime contributions into "MPI-ACC".

MPI-ACC provided a natural interface for programmers to specify actual devices or device abstractions as communication targets, whereas the runtime maps the communication or computation request to the ideal device while maintaining efficient cluster utilization. The data movement and task mapping subsystems within MPI-ACC were designed, not with a one-size-fits-all policy, but with specific tuned optimizations for different cases. The unified programming model ensures that the application programmers have to design their applications once, but will benefit from the evolving optimized runtime.

Our experiments were conducted on *HokieSpeed* – a 212 TFlop CPU-GPU cluster and on *Fire* – an eight-node CPU-GPU cluster, both housed at Virginia Tech. The data movement subsystem of MPI-ACC was evaluated using microbenchmarks and applications from scientific computing domains like seismology and epidemiology. We validated

via the epidemiology application that the MPI-ACC runtime system performed automatic resource management, and was more scalable than the manual MPI+X programming approach. We evaluated our task scheduling system and performance projection model for best device selection by using OpenCL as the example GPU programming model and devices of different generations from both NVIDIA and AMD. Our evaluations on benchmarks and the seismology simulation showed that our model could accurately choose the best device for the given task with minimal error and low overhead.

Bibliography

[1] Heterogeneous System Architecture (HSA). http://www.hsafoundation.com.

[2] MPICH: High-Performance Portable MPI. http://www.mpich.org.

[3] MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE. http://mvapich.cse.ohio-state.edu/.

[4] NVIDIA GPUDirect. https://developer.nvidia.com/gpudirect.

[5] TOP500 Supercomputer Sites. http://www.top500.org/lists/2014/11/highlights.

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

[7] The OpenMP API Specification for Parallel Programming, 2010. http://openmp.org/wp/openmp-specifications.

[8] The OpenACC Application Programming Interface, 2013. http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf.

[9] NVIDIA CUDA C Programming Guide, 2014. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[10] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCToolkit: Tools for Performance Analysis of Optimized Parallel Programs. *Concurrency Computation: Practice and Experience*, 22:685–701, April 2010.

[11] A. M. Aji, P. Balaji, J. Dinan, W.-c. Feng, and R. Thakur. Synchronization and Ordering Semantics in Hybrid MPI GPU Programming. In *3rd Intl. Workshop on Accelerators and Hybrid Exascale Systems (IPDPSW: AsHES)*. IEEE, 2013.

[12] A. M. Aji, M. Daga, and W.-c. Feng. Bounding the Effect of Partition Camping in GPU Kernels. In *ACM International Conference on Computing Frontiers*, Ischia, Italy, May 2011.

[13] A. M. Aji, J. Dinan, D. Buntinas, P. Balaji, W.-c. Feng, K. R. Bisset, and R. Thakur. MPI-ACC: An Integrated and Extensible Approach to Data Movement in Accelerator-Based Systems. In *14th IEEE Intl. Conference on High Performance Computing and Communications (HPCC)*, 2012.

[14] A. M. Aji, L. S. Panwar, F. Ji, M. Chabbi, K. Murthy, P. Balaji, K. R. Bisset, J. Dinan, W.-c. Feng, J. Mellor-Crummy, X. Ma, and R. Thakur. On the Efficacy of GPU-Integrated MPI for Scientific Applications. In *ACM Intl. Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2013.

[15] A. M. Aji, L. S. Panwar, F. Ji, M. Chabbi, K. Murthy, P. Balaji, K. R. Bisset, J. Dinan, W.-c. Feng, J. Mellor-Crummy, X. Ma, and R. Thakur. MPI-ACC: GPU-integrated MPI for Scientific Applications. In *(Submitted) IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2015.

[16] A. M. Aji, L. S. Panwar, J. Meng, P. Balaji, and W.-c. Feng. Online Performance Projection for Clusters with Heterogeneous GPUs. In *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, Seoul, South Korea, December 2013.

[17] A. M. Aji, A. J. Peña, P. Balaji, and W.-c. Feng. Optimizing Task-parallel Workloads Via Automatic Command Queue Scheduling in OpenCL. In *(Submitted) IEEE International Conference on Cluster Computing (CLUSTER)*, 2015.

[18] AMD. APP SDK – A Complete Development Platform. `http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk`.

[19] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[20] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS Parallel Benchmarks. *Intl. Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[21] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[22] C. L. Barrett, K. R. Bisset, S. G. Eubank, X. Feng, and M. V. Marathe. EpiSimdemics: an Efficient Algorithm for Simulating the Spread of Infectious Disease over Large Realistic Social Networks. In *Intl. ACM/IEEE Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2008.

[23] J. Berenger. A Perfectly Matched Layer for the Absorption of Electromagnetic Waves. *Journal of Computational Physics*, 114(2):185–200, 1994.

[24] K. R. Bisset, A. M. Aji, M. V. Marathe, and W.-c. Feng. High-Performance Biocomputing for Simulating the Spread of Contagion over Large Contact Networks. *BMC Genomics*, 13(S2), April 2012.

[25] S. Borkar and A. A. Chien. The Future of Microprocessors. *Communications of the ACM*, 54(5), 2011.

[26] M. Boyer, K. Skadron, and W. Weimer. Automated Dynamic Analysis of CUDA Programs. In *3rd Workshop on Software Tools for MultiCore Systems*, 2010.

[27] W. M. Brown, P. Wang, S. J. Plimpton, and A. N. Tharrington. Implementing Molecular Dynamics on Hybrid High Performance Computers - Short Range Forces. *Computer Physics Communications*, 182(4):898–911, 2011.

[28] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguade, and J. Labarta. Productive Programming of GPU Clusters with OmpSs. In *IEEE Intl. Parallel and Distributed Processing Symposium (IPDPS)*, 2012.

[29] 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.

[30] M. Daga, T. Scogland, and W.-c. Feng. Architecture-Aware Mapping and Optimization on a 1600-Core GPU. In *17th IEEE International Conference on Parallel and Distributed Systems*, December 2011.

[31] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*. ACM, 2010.

[32] C. S. de la Lama, P. Toharia, J. L. Bosque, and O. D. Robles. Static Multi-Device Load Balancing for OpenCL. In *10th International Symposium on Parallel and Distributed Processing with Applications (ISPA)*. IEEE, 2012.

[33] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A Dynamic Compiler for Bulk-Synchronous Applications in Heterogeneous Systems. In *ACM Intl. Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[34] J. Duato, A. J. Peña, F. Silla, R. Mayo, and E. S. Quintana-Orti. rCUDA: Reducing the Number of GPU-based Accelerators in High Performance Clusters. In *Intl. Conference on High Performance Computing and Simulation (HPCS)*. IEEE, 2010.

[35] T. Hamada, T. Narumi, R. Yokota, K. Yasuoka, K. Nitadori, and M. Taiji. 42 TFlops Hierarchical N-body Simulations on GPUs with Applications in both Astrophysics and Turbulence. In *ACM/IEEE Intl. Conf. for High Perf. Computing, Networking, Storage and Analysis (SC)*, 2009.

[36] S. Henry, A. Denis, D. Barthou, M.-C. Counilh, and R. Namyst. Toward OpenCL Automatic Multi-Device Support. In *Euro-Par Parallel Processing*. Springer, 2014.

[37] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. Gropp, V. Kale, and R. Thakur. MPI+MPI: A New Hybrid Approach to Parallel Programming with MPI Plus Shared Memory. *Computing*, 95(12):1121–1136, 2013.

[38] S. Hong and H. Kim. An Analytical Model for a GPU Architecture with Memory-Level and Thread-Level Parallelism Awareness. In *ACM International Symposium on Computer Architecture (ISCA)*. ACM, 2009.

[39] L. Howes and A. Munshi. *The OpenCL Specification*. Khronos OpenCL Working Group, 2014. `https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf`.

[40] IBM. *OpenCL Common Runtime for Linux on x86 Architecture (Version 0.1)*, 2011.

[41] F. Ji, A. M. Aji, J. Dinan, D. Buntinas, P. Balaji, W.-c. Feng, and X. Ma. Efficient Intranode Communication in GPU-Accelerated Systems. In *2nd Intl. Workshop on Accelerators and Hybrid Exascale Systems (IPDPSW: AsHES)*. IEEE, 2012.

[42] F. Ji, A. M. Aji, J. Dinan, D. Buntinas, P. Balaji, R. Thakur, W.-c. Feng, and X. Ma. DMA-Assisted, Intranode Communication in GPU Accelerated Systems. In *14th IEEE Intl. Conference on High Performance Computing and Communications (HPCC)*, 2012.

[43] Kaixi Hou. FDM-Seismology in OpenCL. Personal Copy.

[44] R. Kaleem, R. Barik, T. Shpeisman, B. T. Lewis, C. Hu, and K. Pingali. Adaptive Heterogeneous Scheduling for Integrated GPUs. In *23rd ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2014.

[45] J. Kim, H. Kim, J. H. Lee, and J. Lee. Achieving a Single Compute Device Image in OpenCL for Multiple GPUs. In *16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2011.

[46] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee. SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters. In *26th ACM International Conference on Supercomputing (ICS)*, 2012.

[47] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. PyCUDA and PyOpenCL: A Scripting-based Approach to GPU Run-time Code Generation. *Parallel Computing*, 38(3):157–174, 2012.

[48] P. Lama, Y. Li, A. M. Aji, P. Balaji, J. Dinan, S. Xiao, Y. Zhang, W.-c. Feng, R. Thakur, and X. Zhou. pVOCL: Power-Aware Dynamic Placement and Migration in Virtualized GPU Environments. In *International Conference on Distributed Computing Systems (ICDCS)*, 2013.

[49] O. S. Lawlor. Message Passing for GPGPU Clusters: cudaMPI. In *IEEE Intl. Conference on Cluster Computing and Workshops (CLUSTER)*, 2009.

[50] Lawrence Livermore National Laboratory. SLURM Generic Resource (GRES) Scheduling. `https://computing.llnl.gov/linux/slurm/gres.html`, 2012.

[51] B. C. Lee and D. M. Brooks. Accurate and Efficient Regression Modeling for Micro-architectural Performance and Power Prediction. In *ACM Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[52] C.-K. Luk, S. Hong, and H. Kim. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. In *42nd IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.

[53] 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.

[54] 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.

[55] J. Meng and K. Skadron. Performance Modeling and Automatic Ghost Zone Optimization for Iterative Stencil Loops on GPUs. In *23rd ACM International Conference on Supercomputing (ICS)*, 2009.

[56] A. Nere, A. Hashmi, and M. Lipasti. Profiling Heterogeneous Multi-GPU Systems to Accelerate Cortically Inspired Learning Algorithms. In *IEEE Intl. Parallel and Distributed Processing Symposium (IPDPS)*, 2011.

[57] P. Pandit and R. Govindarajan. Fluidic Kernels: Cooperative Execution of OpenCL Programs on Multiple Heterogeneous Devices. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2014.

[58] L. S. Panwar, A. M. Aji, J. Meng, P. Balaji, and W.-c. Feng. Online Performance Projection for Clusters with Heterogeneous GPUs. In *19th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2013.

[59] PBS Works™. Scheduling Jobs onto NVIDIA Tesla GPU Computing Processors using PBS Professional. Technical report, 2010.

[60] A. J. Peña. *Virtualization of Accelerators in High Performance Clusters*. PhD thesis, Universitat Jaume I (Spain), 2013.

[61] Pengcheng Liu. DISFD in Fortran. Personal Copy.

[62] J. Phillips, J. Stone, and K. Schulten. Adapting a Message-Driven Parallel Application to GPU-accelerated Clusters. In *ACM/IEEE Intl. Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2008.

[63] V. T. Ravi, M. Becchi, W. Jiang, G. Agrawal, and S. Chakradhar. Scheduling Concurrent Applications on a Cluster of CPU–GPU Nodes. *Future Generation Computer Systems*, 2013.

[64] G. Ruetsch and P. Micikevicius. Optimizing Matrix Transpose in CUDA. `http://docs.nvidia.com/cuda/samples/6_Advanced/transpose/doc/MatrixTranspose.pdf`.

[65] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-m. W. Hwu. Program Optimization Space Pruning for a Multithreaded GPU. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2008.

[66] T. Scogland, W.-c. Feng, B. Rountree, and B. R. de Supinski. CoreTSAR: Adaptive Worksharing for Heterogeneous Systems. In *Supercomputing*, pages 172–186. Springer International Publishing, 2014.

[67] T. Scogland, B. Rountree, W.-c. Feng, and B. R. de Supinski. Heterogeneous Task Scheduling for Accelerated OpenMP. In *IEEE Intl. Parallel and Distributed Processing Symposium (IPDPS)*, 2012.

[68] S. Seo, G. Jo, and J. Lee. Performance Characterization of the NAS Parallel Benchmarks in OpenCL. In *IEEE Intl. Symposium on Workload Characterization (IISWC)*, 2011.

[69] T. Shimokawabe, T. Aoki, C. Muroi, J. Ishida, K. Kawano, T. Endo, A. Nukada, N. Maruyama, and S. Matsuoka. An 80-Fold Speedup, 15.0 TFlops Full GPU Acceleration of Non-Hydrostatic Weather Model ASUCA Production Code. In *ACM/IEEE Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.

[70] M. Si, A. J. Peña, P. Balaji, M. Takagi, and Y. Ishikawa. MT-MPI: Multithreaded MPI for Many-core Environments. In *28th ACM International Conference on Supercomputing (ICS)*, 2014.

[71] J. Sim, A. Dasgupta, H. Kim, and R. 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.

[72] K. Spafford, J. Meredith, and J. Vetter. Maestro: Data Orchestration and Tuning for OpenCL Devices. In *Euro-Par – Parallel Processing*, pages 275–286. Springer, 2010.

[73] Steve Rennich. CUDA C/C++ Streams and Concurrency. `http://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf`.

[74] J. Stuart and J. Owens. Message Passing on Data-Parallel Architectures. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2009.

[75] V. Taylor, X. Wu, and R. Stevens. Prophesy: An Infrastructure for Performance Analysis and Modeling of Parallel and Grid Applications. *SIGMETRICS Performance Evaluation Review*, 30(4), 2003.

[76] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. Multi2Sim: A Simulation Framework for CPU-GPU Computing . In *ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

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

[78] H. Wang, S. Potluri, M. Luo, A. Singh, S. Sur, and D. Panda. MVAPICH2-GPU: Optimized GPU to GPU Communication for InfiniBand Clusters. *International Supercomputing Conference (ISC)*, 2011.

[79] L. 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.

[80] Y. Wen, Z. Wang, and M. O'Boyle. Smart Multi-Task Scheduling for OpenCL Programs on CPU/GPU Heterogeneous Platforms. In *21st IEEE International Conference on High Performance Computing (HiPC)*, 2014.

[81] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU Microarchitecture Through Microbenchmarking. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2010.

[82] S. Xiao, P. Balaji, J. Dinan, R. Thakur, S. Coghlan, H. Lin, G. Wen, J. Hong, and W.-c. Feng. Transparent Accelerator Migration in a Virtualized GPU Environment. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2012.

[83] S. Xiao, P. Balaji, Q. Zhu, R. Thakur, S. Coghlan, H. Lin, G. Wen, J. Hong, and W.-c. Feng. VOCL: An Optimized Environment for Transparent Virtualization of Graphics Processing Units. In *IEEE Innovative Parallel Computing (InPar)*, 2012.

[84] Y. Zhang and J. D. Owens. A Quantitative Performance Analysis Model for GPU Architectures. In *ACM International Symposium on High-Performance Computer Architecture (HPCA)*, 2011.