University of Kentucky
UKnowledge

University of Kentucky Master's Theses | Graduate School

2009

MPI WITHIN A GPU

Bobby Dalton Young
University of Kentucky, bdyoun0@engr.uky.edu

Right click to open a feedback form in a new tab to let us know how this document benefits you.

Recommended Citation

Young, Bobby Dalton, "MPI WITHIN A GPU" (2009). *University of Kentucky Master's Theses*. 614.
https://uknowledge.uky.edu/gradschool_theses/614

This Thesis is brought to you for free and open access by the Graduate School at UKnowledge. It has been accepted for inclusion in University of Kentucky Master's Theses by an authorized administrator of UKnowledge. For more information, please contact UKnowledge@lsv.uky.edu.

ABSTRACT OF THESIS

MPI WITHIN A GPU

GPUs offer high-performance floating-point computation at commodity prices, but their usage is hindered by programming models which expose the user to irregularities in the current shared-memory environments and require learning new interfaces and semantics.

This thesis will demonstrate that the message-passing paradigm can be conceptually cleaner than the current data-parallel models for programming GPUs because it can hide the quirks of current GPU shared-memory environments, as well as GPU-specific features, behind a well-established and well-understood interface. This will be shown by demonstrating a proof-of-concept MPI implementation which provides cleaner, simpler code with a reasonable performance cost. This thesis will also demonstrate that, although there is a virtualization constraint imposed by MPI, this constraint is harmless as long as the virtualization was already chosen to be optimal in terms of a strong execution model and nearly-optimal execution time. This will be demonstrated by examining execution times with varying virtualization using a computationally-expensive micro-kernel.

KEYWORDS: message-passing, virtualization, data-parallel, virtualization MPI, GPU

Bobby Dalton Young

August 4, 2009

MPI WITHIN A GPU

By

Bobby Dalton Young

Henry G. Dietz, Ph.D.
Director of Thesis

Stephen D. Gedney, Ph.D.
Director of Graduate Studies

August 4, 2009
Date

RULES FOR THE USE OF THESES

Unpublished theses submitted for the Master's degree and deposited in the University of Kentucky Library are as a rule open for inspection, but are to be used only with due regard for the rights of the authors. Bibliographical references may be noted, but quotations or summaries of parts may be published only with the permission of the author, and with the usual scholarly acknowledgments.

Extensive copying or publication of the thesis in whole or part also requires the consent of the Dean of the Graduate School of the University of Kentucky.

A library that borrows this thesis for use by its patrons is expected to secure the signature of each user.

Name	Date

THESIS

Bobby Dalton Young

The Graduate School

University of Kentucky

2009

MPI WITHIN A GPU

THESIS

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science in the
College of Engineering
at the University of Kentucky

By

Bobby Dalton Young

Lexington, Kentucky

Director: Dr. Henry G. Dietz, Professor of Electrical Engineering

Lexington, Kentucky

2009

Copyright © Bobby Dalton Young 2009

TABLE OF CONTENTS

List of Tables....iv

List of Figures....v

List of Files....vi

Chapter 1: Introduction....1

Chapter 2: Background, Methodology, and Related Work....2
- 2.1: Background....2
- 2.2: Methodology....4
- 2.3: Related Work....5

Chapter 3: GPU Hardware Review and Performance Factors....7
- 3.1: Review of Current GPU Hardware....7
- 3.2: The NVIDIA CUDA Architecture....9
- 3.3: NVIDIA CUDA Performance Considerations....12

Chapter 4: Virtualization Constraints....15
- 4.1: Background and Definitions....15
- 4.2: Performance Analysis of Virtualization....17
- 4.3: Optimal Virtualization and the MPI Constraints....26

Chapter 5: The MPI Implementation....27
- 5.1: Introduction to the Message-Passing Interface....27
- 5.2: Design Philosophies and Restrictions....28
- 5.3: The Point-to-Point Communication Interfaces....33
- 5.4: Point-to-Point Communication Performance....42
- 5.5: The Collective Communication Interfaces....48
- 5.6: Collective Communication Performance....59
- 5.7: Other Implemented Interfaces....65
- 5.8: Costs and Benefits of the Message-Passing Model....67

Chapter 6: Conclusions and Future Work....68

Appendix A: CUDA Code for the Functions....69

Bibliography....85

Vita....89

LIST OF TABLES

Table 3.1: Some NVIDIA GPUs and various properties.....10
Table 3.2: Compute Capabilities and their properties[23].....11

LIST OF FIGURES

Figure 3.1: The NVIDIA CUDA Architecture[23].....10
Figure 4.1: Pseudo-code for the micro-benchmark.....18
Figure 4.2: The register micro-benchmark.....19
Figure 4.3: Total execution time of the register micro-benchmark.....20
Figure 4.4: Per-block execution time of the register micro-benchmark.....20
Figure 4.5: The shared-memory micro-benchmark.....22
Figure 4.6: Total execution time of the shared-memory micro-benchmark.....23
Figure 4.7: Per-block execution time of the shared-memory micro-benchmark.....23
Figure 5.1: MPI_Send prototype.....34
Figure 5.2: Pseudo-code for MPI_Send.....35
Figure 5.3: MPI_Recv Prototype.....37
Figure 5.4: Pseudo-code for MPI_Recv.....38
Figure 5.5: Algorithm for sequential estimation of PI.....43
Figure 5.6: Point-to-point MPI implementation of the test algorithm.....44
Figure 5.7: Point-to-point CUDA-only implementation of the test algorithm, part 1.....45
Figure 5.8: Point-to-point CUDA-only implementation of the test algorithm, part 2.....46
Figure 5.9: Execution time of both implementations of the test algorithm.....47
Figure 5.10: MPI_Barrier prototype.....49
Figure 5.11: Pseudo-code for MPI_Barrier.....50
Figure 5.12: MPI_Reduce prototype macro.....52
Figure 5.13: MPI_Reduce prototype.....52
Figure 5.14: MPI_Reduce translation macro.....53
Figure 5.15: Pseudo-code for MPI_Reduce.....54
Figure 5.16: Algorithm for sequential estimation of PI.....59
Figure 5.17: Collective MPI implementation of the test algorithm.....61
Figure 5.18: Collective CUDA-only implementation of the test algorithm, part 1.....62
Figure 5.19: Collective CUDA-only implementation of the test algorithm, part 2.....63
Figure 5.20: Execution time of both implementations of the test algorithm.....64

LIST OF FILES

BDYthesis.pdf 522KB

Chapter 1: Introduction

GPUs have gained attention for their impressive floating-point price/performance ratio. The consumer demand for realism and resolution in traditional graphics applications has effectively created scalable, high-performance floating-point hardware at commodity prices. GPUs have already been used to accelerate some numerically-intensive high-performance computing applications, but limitations imposed by current GPU programming methodologies, including a lack of code portability between GPU platforms, have hindered the use of GPUs in a wider class of programs and relegated them to a second-class co-processor status.

This thesis aims primarily to demonstrate that the message-passing programming model can be conceptually cleaner than the vendor-supplied data-parallel models by hiding the quirks of current shared-memory environments. This is accomplished by demonstrating a proof-of-concept implementation of a message-passing model on a GPU and providing performance comparisons between two GPU programs, one of which utilizes the message-passing implementation and the other of which does not, on a parallel test algorithm computing the value of pi. The message-passing implementation will be a subset of MPI[1] (the Message Passing Interface), which is a popular message-passing library specification, and the implementation and test algorithms will be developed and tested in NVIDIA CUDA[2] (the Compute Unified Device Architecture), which is NVIDIA's parallel architecture and model for general-purpose computing on a GPU.

This thesis also aims to demonstrate that, although there is a virtualization constraint for using message-passing (and more specifically MPI) within the GPU, the constraint is actually harmless if the amount of virtualization chosen is already optimal in terms of its interaction model and per-block execution time. This is accomplished by ignoring the amount of virtualization needed by MPI and profiling different amounts of virtualization on a simple micro-kernel benchmark. The resulting execution times demonstrate that the amount of virtualization needed to obtain the optimal performance, with optimal performance as defined above, is compatible with the MPI virtualization constraint.

The rest of this thesis is structured as follows. Chapter 2 (Background,

Methodology, and Related Work) provides a brief review of the message-passing and data-parallel programming paradigms discussed above, describes the relevant classes of architectures and how they apply to GPUs, describes an important philosophy which underlies this thesis, and cites related work. Chapter 3 (GPU Hardware Review and Performance Factors) provides a hardware overview which compares current-generation GPU architectures and then provides an in-depth review of the NVIDIA CUDA architecture and important performance considerations for it. Chapter 4 (Virtualization Constraints) discusses the ideas of virtualization, multi-threading, global-memory coherence, and interaction models, presents empirical data regarding the optimum amount of multi-threading and virtualization, and finally examines how the virtualization constraints imposed by MPI within a GPU are satisfied at the optimum amount of virtualization. Chapter 5 (The MPI Implementation) introduces the Message Passing Interface standard, discusses some design philosophies underlying the proof-of-concept implementation, and examines key function implementations which provide point-to-point and aggregate communication operations. Chapter 5 also provides performance comparisons between canonical NVIDIA CUDA and the MPI implementation for two implementations of an algorithm computing the value of pi in parallel and discusses the costs and benefits of using the message-passing implementation. Chapter 6 (Conclusions and Future Work) presents conclusions and future work for the research.

Chapter 2: Background, Methodology, and Related Work

This section first defines the data-parallel and message-passing paradigms and discusses how they apply to the thesis. This section also defines SIMD, SPMD, SIMT, and MIMD, since these architecture classifications are used extensively in the thesis. The overarching philosophies of this thesis are then discussed after clarifying the definitions and applications of these background concepts. Finally, related work is discussed.

2.1: Background

Two of the popular paradigms for programming parallel machines are the data-parallel paradigm and the message-passing paradigm. These programming models are described in detail below.

The data-parallel paradigm, exemplified by HPF[3] and current GPU programming environments such as CUDA[2], BrookGPU[4], and CAL[5], typically consists of a single program controlling execution of all processing elements. In this paradigm, program data is typically stored in arrays, and processing elements work on subsets of arrays in their local memories.

Historically, GPU programming methods and environments have been based on the data-parallel paradigm. This is largely due to the fact that GPUs have evolved to process pixels of an image in parallel. Maintaining the same paradigm for general-purpose programming has not been widely questioned, largely because the majority of GPU users are still performing graphics-processing applications (as opposed to general-purpose computation). While general-purpose computation on GPUs is gaining popularity, it is still a small portion of the total market share of current GPUs. While this has allowed GPU vendors to both continue normal product development cycles and broaden marketability by advertising GPUs for non-graphics applications, it has restricted the use of the GPU to programs where data-parallelism is easily exposed. If GPUs can support the message-passing paradigm in addition to the data-parallel paradigm, though, then a wider range of execution models can be supported on current GPUs.

Message-passing is not the opposite of data-parallelism; the opposite is actually task-parallelism. This thesis will show that message-passing is compatible with data-parallelism. The message-passing paradigm, exemplified by the MPI standard[1] and various implementations of it such as OpenMPI[6] and LAM-MPI[7], typically consists of individual programs on each processing element controlling local execution. In this paradigm, program data is typically passed between processing elements via cooperative send and receive mechanisms.

Before continuing with the idea of message-passing on a GPU, some basic architectural ideas must be mentioned. Parallel computer architectures have long been classified as SIMD (Single-Instruction stream, Multiple-Data stream) and MIMD (Multiple-Instruction stream, Multiple-Data stream)[8]. As the names imply, SIMD hardware typically consists of a collection of processing elements, sometimes little more than Arithmetic Logic Units, all executing a single instruction on multiple data. MIMD hardware, in contrast, typically consists of processing elements with their own

instruction-fetch, decode, and execute logic, each executing a potentially different instruction on multiple data.

In addition to the SIMD and MIMD classifications, there are also SPMD systems, and the term "SIMT" should be discussed since NVIDIA uses it. SPMD (Single Program, Multiple-Data stream) are akin to SIMD systems, except that the processing elements are potentially much less synchronized because they are executing programs rather than instructions[9]. These are also akin to MIMD systems where every processing element is executing an identical program. SIMT is a term coined by NVIDIA, and stands for Single-Instruction, Multiple-Thread[2]. While it attempts to describe the scheduling of threads to available processor cores, it can best be thought of as SIMD with virtualization at the thread level.

GPUs have traditionally been classified as SIMD-like or SPMD-like architectures. As the hardware review section will illustrate, though, GPUs are a strange hybrid of the two, and behave differently at different granularities. This plays a role in enabling the implementation of a message-passing system. More importantly, though, the multiple potential classifications of GPU architectures can and should be leveraged to further expand the set of execution models which are viable within a GPU.

2.2: Methodology

In chapter 1, the idea of the GPU being relegated to an attached co-processor role was mentioned. The majority of GPU research approaches the GPU as a second-class processing element and utilizes the GPU as an attached accelerator, but this thesis approaches the GPU as a future first-class parallel processor, which motivates the idea of needing a message-passing system within a GPU. If the GPU will internally be a separate entity, then it will need to perform synchronization and communication without assistance. The work in this thesis is not merely about using more execution models within a GPU, it is about which models will be needed as GPUs become independent.

We envision the use of the GPU taking the same path as the use of traditional floating-point co-processors, such as the Intel 8087 floating-point co-processor. While floating-point math was originally only available via an attached co-processor accessed with OS calls, it is now an integral part of the chip design and instruction set. Intel's Larrabee, scheduled for a 2009 or 2010 release, will be an explicitly standalone GPU

with many CPU features, including x86 processing cores[10]. Additionally, Larrabee's SIMD extensions will be opcode-compatible with the specification for extending SSE[10]. The Cell processor already includes SIMD SPEs (Synergistic Processing Elements) which work alongside a Power-architecture based PPE (Power Processing Element)[11]. AMD's FUSION project originally proposed "Accelerated Processing Units" (CPUs with a GPU on-die) as early as 2007[12], and these are now scheduled for a 2011 release in the Llano and Ontario cores[13].

Since this thesis holds that the models for GPUs are converging to this single environment, the performance benchmarks in this thesis are focused on examining performance in the context of a self-contained world, with the GPU behaving as normal computing systems generally behave. Since there is not much existing work with this viewpoint to benchmark performance against, and since benchmarking performance against a sequential code would be unfair, the benchmarks in this thesis focus on describing and quantifying the performance penalty or cost for the more-general model provided by MPI. Hopefully, future GPU research will yield competing models so that better performance comparisons can be accurately made.

2.3: Related Work

There is considerable interest in GPGPU (General-Purpose Computation on GPUs), which generally refers to using graphics cards to perform computations which are not strictly limited to manipulating graphics on the screen. Although this work is related to many GPGPU efforts described below, note that this work is focused on developing a more-general programming model within the actual GPU. As stated in chapter 2.2, this research is aimed at developing a first-class processing element.

Early GPGPU programming revolved around graphics programming languages such as DirectX[14] and OpenGL[15]. These languages developed programmable shader pipelines, controlled by the HLSL[16] and GLSL[17] languages, and programmers used the shader languages to process non-graphical data. Some higher-level languages, such as Cg[18], were also used. Because of the awkwardness of expressing non-graphical computations in graphically-oriented languages, general-purpose languages which could hide the underlying nature of the hardware and programming model began to emerge. These included Sh[19], which made the GPU hardware appear as a simple resource via a

C++ library, and BrookGPU[4], which focused on programming in streams with a C++ dialect. Microsoft research also worked on an advanced system called Accelerator[20], which performed advanced just-in-time compilation with an OpenGL backend. Some companies also provided proprietary solutions, such as RapidMind's Multi-Core Platform[21].

As GPGPU became popular, vendors began to provide languages which allowed better access to the underlying hardware. ATI released CTM[22], which included the assembly language of the GPUs and driver-level codes to operate them. NVIDIA released CUDA[23], which provided a general-purpose abstraction of the underlying hardware, but did not release instruction set architectures. The development of these languages also lead to the development of more companies providing GPU acceleration services, such as PeakStream[24]. ATI's CTM was later obsoleted in favor of CAL[25], which provides similar low-level access to CTM, but with a modified version of the BrookGPU compiler which allowed users to avoid low-level programming. Efforts are currently underway to develop a vendor-independent specification for a GPGPU programming language called OpenCL[26].

Currently, many research efforts related to GPU programming environments are underway. CUDASA[27] is a system for networking clusters of machines with GPUs together, and it extends CUDA with new language constructs to do so. Zippy[28] provides similar functionality via library routines and classes which abstract the underlying GPUs. More recently, DCGN[29] abstracted GPUs in clusters, but allowed dynamic communication patterns during execution between GPUs. All of these efforts are related to using message-passing across clusters of GPUs and abstracting the hardware, whereas this research is interested in exploring the message-passing model within the hardware.

The most closely-related work is actually the BSGP project[30], which focuses on a set of extensions and very advanced compiler techniques to transform and execute sequential code on a GPU. BSGP still lacks inter-thread communication, though, and implements such mechanisms by generating individual kernels and using the edges as synchronization points. This mechanism still relies on a host processor, and still leaves the GPU as a second-class processing element.

Although virtualization is not the main focus of this research, it is worth noting that virtualization in SIMD machines has been well studied before. The PARIS language[31] utilized by Thinking Machines SIMD computers is a good example.

Chapter 3: GPU Hardware Review and Performance Factors

To understand the proof-of-concept MPI implementation and the virtualization test results, it is necessary to understand the underlying CUDA hardware. This section first provides a high-level overview of current GPU hardware, which demonstrates the applicability of this work to other GPUs beyond just NVIDIA targets. Next, it provides an overview of the CUDA hardware and the scaling or scheduling parameters used to control the hardware. This section concludes with a discussion of performance factors within an NVIDIA CUDA GPU; this information needed to understand the virtualization test results in chapter 4 and some design decisions used in the MPI implementation in chapter 5.

3.1: Review of Current GPU Hardware

Modern GPUs are composed of a collection of virtualizing SIMD processors. Note that these are not multiprocessors, since they do not support multiprocessing. These are relatively narrow virtualizing SIMD engines which support multi-threading over a set of processing elements, and support no MIMD behavior aside from virtualization and multi-threading.

The individual virtualizing SIMD processors are composed of some number of physical processing elements, each with its own registers and all sharing some local shared memory. Each virtualizing SIMD processor possesses local control logic for its processing elements, and all SIMD processors in a GPU can execute a different instruction at the same time. All processing elements on the SIMD processors can access a global shared memory, and all SIMD processors are connected through some global control and scheduling logic. The processing elements within a single SIMD processor can also access a local shared memory inside that multiprocessor.

The smallest unit of work in a modern GPU is a sequence of instructions executing on a single processing element. Note that this unit of work is not a thread in

the traditional sense (although the word has been appropriated to describe it), but rather a "slice" of a traditional thread executing a SIMD program. A thread in the traditional sense is either a group of work units which execute simultaneously on the processing elements or a group of work units which form a single schedulable entity.

The units of work are typically grouped at two granularities visible to the user. At the smaller granularity, some number of parallel work units form a single schedulable unit, with the size of the group usually determined by the number of logical processing cores inside a SIMD engine (which may be more than the number of physical cores). At the larger granularity, the aforementioned groups which are executed together on a single SIMD processor are grouped again. In some sense, the architecture is MIMD at the device or higher granularity level, since SIMD processors can execute unique instructions simultaneously; and virtualizing SIMD or SPMD at the SIMD processor or lower granularity level, since instruction execution is overlapped and scheduled on the processing cores.

From reviewing the NVIDIA CUDA Programming Guide[23], the above concepts are clearly seen in NVIDIA GPUs. In a CUDA GPU, the virtualizing SIMD processors are the SMs (Streaming Multiprocessors). Each SM consists of eight physical SP (Scalar Processor) cores executing 32 work units (called threads) in four clock cycles, and all SPs in an SM have access to a shared memory. Groups of 32 work units are called warps (the smaller granularity and smallest schedulable entity), and their execution is scheduled to hide memory latency. The work units are grouped into what NVIDIA calls thread blocks (the larger granularity, inside of which execution is virtualizing SIMD or SPMD) which are distributed to the SMs by some global control logic, and multiple thread blocks can execute on a single SM. Between SMs, divergence in code causes no performance penalty and is effectively MIMD. As work units execute within an SM, only divergence (due to control flow instructions) within a warp is implemented by masking and serialization; divergence between warps is implemented by scheduling from the local control logic.

From reviewing the ATI Stream Computing User Guide[22], the concepts are also seen in ATI GPUs. In a CAL GPU, the virtualizing SIMD processors are called SIMD engines. The SIMD engines are composed of what ATI calls thread processors, with all

thread processors executing the same instruction in unison. Work units (again called threads) are grouped into wavefronts (the smaller granularity and smallest schedulable entity), with the size of a wavefront determined by the number of thread processors times 4, since each thread processor can issue 4 instructions in 4 clock cycles. All thread processors within a SIMD engine share a LDS (Local Data Store) shared memory (which is owner-writes) as well as shared registers which can communicate data between wavefronts. The work units executed on the processors are arranged into groups at the software level (the larger granularity, inside of which execution is virtualizing SIMD or SPMD), and multiple wavefronts are overlapped in each SIMD engine to hide memory latency. During execution, only divergence within a wavefront is implemented by thread masking and serialization. Divergence between SIMD engines has no performance penalty, and the execution is effectively MIMD at this granularity.

From the above descriptions, it should be obvious that the results obtained using an NVIDIA GPU are applicable to other GPUs. This is not to say that no differences exist, especially since the next few sections on hardware and performance apply mainly to NVIDIA GPUs. The differences between these architectures, however, are minor, and any impacts they would be expected to cause in the implementation are noted as they arise in the thesis.

Note that the rest of this thesis will use NVIDIA terminology (threads instead of work units, and SMs instead of SIMD processors), since the research utilizes an NVIDIA CUDA GPU.

3.2: The NVIDIA CUDA Architecture

Now, an overview of the NVIDIA CUDA architecture is needed. Figure 3.1 shows a simple overview of hardware used by a CUDA program in an NVIDIA GPU. All NVIDIA GPUs share a similar hardware structure, but may have different clock rates, SM counts, shared memory sizes, etc. Table 3.1 shows some of these differences for the NVIDIA GPUs used in this work.

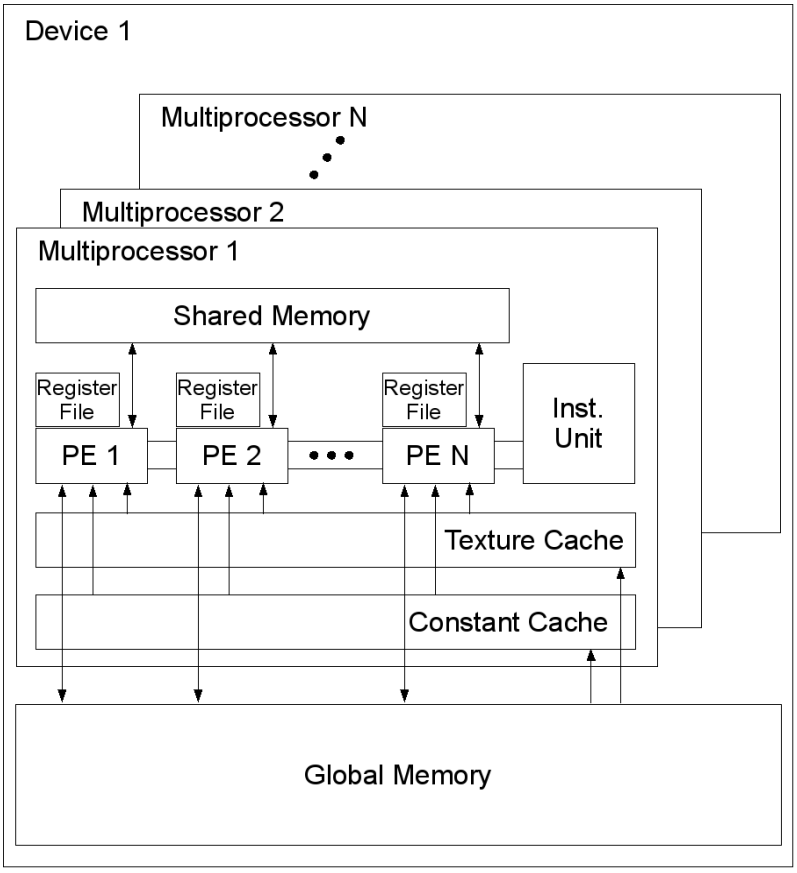

Figure 3.1: The NVIDIA CUDA Architecture[23]

Table 3.1: Some NVIDIA GPUs and various properties

GeForce GPU	8800 GT[32]	9800 GT[33]	280 GTX[34]
Number of SMs	14	16	30
Number of SPs	112	128	240
Device Global Mem	512 MB	512 MB – 1 GB	1 GB
Compute Capability	1.1	1.1	1.3
Processor Clock	1500 MHz	1500 MHz	1296 MHz
Memory Clock	900 MHz	900 MHz	1107 MHz

As can be seen in Figure 3.1, the device architecture is composed of a collection of virtualizing SIMD processors (the Streaming Multiprocessors, or SMs), all of which share access to a global device memory. Each SM is composed of SIMD processing elements (the Scalar Processors, or SPs) which execute instructions issued by the

instruction unit. Each SP possesses a separate register file, and all SPs in a block share access to a multi-banked shared memory, texture cache, and constant cache. The sizes of the various memories and caches, as well as the register and SP counts, vary with what NVIDIA terms the "compute capability" of the device. Specifications for the various compute capabilities are shown in Table 3.2. Note that the device used for the research in this thesis is a compute-capability 1.0 device, which is the most restrictive of the compute capabilities. This was necessary to guarantee that the results in this thesis are portable.

Table 3.2: Compute Capabilities and their properties[23]

Compute Capability	1.0 & 1.1	1.2 & 1.3
Physical PEs per SM	8	8
Constant Memory	64k	64k
Shared Memory per SM	16k	16k
Registers per SM	8192	16384
Constant Cache per SM	8k	8k
Texture Cache per SM	6k - 8k	6k - 8k
Warp Size	32	32
Max Threads per Block	512	512
Max Blocks per SM	8	8
Max Warps per SM	24	32
Max Threads per SM	768	1024

In this system, what NVIDIA calls threads are the smallest unit of work. Each thread executes a copy of the kernel, and groups of threads are scheduled in a SIMD fashion on the SPs inside the individual SMs. More specifically, the eight physical SPs in current CUDA devices appear as 32 logical SPs to the scheduler, and each physical SP executes an instruction from each of four threads in four clock cycles. This appears logically as 32 threads executing one instruction every four clock cycles. A group of 32 threads is called a "warp" in NVIDIA terminology, and each SM overlaps execution of the warps of 32 threads on the SPs in order to hide memory fetch latency.

Warps of threads are grouped into thread blocks (commonly referred to as blocks), and thread blocks are the basic element of virtualization. A single SM may execute many

blocks at once (up to a compute-capability defined limit, as shown in Table 3.2), but only if the combined thread count of the blocks does not exceed the SM's limit, and the resource requirements of the blocks can be met. Resource requirements for a block include the amount of shared memory and the number of registers used by the threads in the block, and the compute capability of the device determines the available resources. For example, if an SM will overlap execution of 4 thread blocks, then it must possess at least 4 times the total resources needed by each thread block.

Scheduling parameters (or scaling parameters, since they tightly constrain the number of kernels active at once on a GPU) govern the creation of thread blocks, and are provided by the user when a kernel program is invoked in host CPU code to execute on the GPU hardware. These parameters include the dimensions of a thread block (in threads), the dimensions of the grid (in thread blocks), and the amount of dynamically shared memory allocated per thread block. NVIDIA calls this collection of parameters the "execution configuration" of a kernel. These parameters are often chosen by the user to allow easy mapping of the desired algorithm to the available GPU resources.

If more thread blocks are created by a kernel invocation than can be executed on the GPU's SMs, due either to a lack of resources or to more combined threads than the SMs can execute, some thread blocks are queued to wait. These thread blocks do not execute until another thread block has executed to completion. Lacking provision to save and restore state, there is no `yield()` or `sleep()` command which a block can issue to voluntarily de-schedule itself, although one would be extremely useful.

3.3: NVIDIA CUDA Performance Considerations

Now that all the fundamentals are established, it is important to discuss the key performance characteristics in the CUDA architecture. These are essentially the points from chapter five of the NVIDIA CUDA Programming Guide[23], and they concern instruction performance and memory performance.

Instruction performance falls loosely into three categories: arithmetic performance, flow-control performance, and memory instruction performance. By examining each, a scale of relative expense should be established in the reader's mind, and this scale is required to understand design decisions described later in this thesis.

The shortest arithmetic instructions execute in four clock cycles, and these include

floating-point add, multiply, multiply-add, integer add, 24-bit integer multiply, bitwise operations, minimum, maximum, comparison, and type-conversion instructions. 32-bit integer multiply executes in 16 clock cycles, and floating-point reciprocal square root executes in 32 clock cycles. Floating-point division executes in 20 or 36 clock cycles (depending on the version used), and some versions of sin, cos, and exponentiation execute in 32 clock cycles. The instructions which are specified with the longest execution time are other versions of sin and cos, tan, and sincos, which may take around 320 clock cycles to execute.

Flow-control instructions are not specified with execution times, and are primarily important in that divergent code can cause serious performance degradation or deadlock. In particular, divergence between warps is not problematic, because it is implemented by scheduling. Within a warp, though, divergence is implemented by masking off processing elements and serializing the branches. The performance penalty can be high in such instances. An example: if a single processing element in each warp wished to execute instructions requiring 10,000 clock cycles, while all other processing elements in the warp wished to execute instructions requiring 10 clock cycles, the total execution time of each warp would be 10,010 clock cycles. Worse still is the possibility of live-lock. If a processing element inside of a flow-control statement which diverged within a warp attempts a blocking operation (such as a spin-lock) while waiting for other processing elements in the warp to signal a result, the result may never arrive due to the masking-off of the processing elements which send it, and the hardware may live-lock until the watchdog timer aborts the kernel execution.

Memory instructions are simple in that they always issue in 4 clock cycles. The problem, as shown next, is the memory latency.

Memory performance in a CUDA GPU is the single most important performance consideration short of outright deadlock. A fetch from global or local memory locations has a 400-600 clock cycle latency, and global memory is not cached. Shared memory accesses which are fully-coalescing and conflict-free (across the banks) have a four-cycle latency which is hidden by the warp-at-a-time scheduling. Similarly, dependency-free register accesses have a four-cycle latency hidden by the scheduling. Texture and constant cache reads are also four-cycle unless the read is a cache miss, and cache misses

incur the 400-600 clock cycle global memory latency.

Memory performance can be improved considerably by arranging memory accesses to be coalesced. As described in the NVIDIA CUDA Programming Guide[23], CUDA devices are capable of fetching 32-byte, 64-byte, and 128-byte blocks aligned on 32-byte, 64-byte, and 128-byte boundaries respectively, in single instructions. This occurs only under certain conditions based on the device compute-capability and for each half-warp (group of 16 consecutive threads). For compute-capability 1.0 and 1.1 devices (such as the GeForce 8800 GTS used in this thesis), conditions for coalescing include threads accessing consecutive words in sequence, with each access being a 4-byte or 8-byte object (16-byte object accesses are coalesced into two 128-byte load instructions). It should be noted that coalesced accesses are around an order of magnitude higher bandwidth than non-coalesced on 4-byte objects, around four times higher for 8-byte objects, and around two times higher for 16-byte objects. On compute-capability 1.2 or higher devices, coalescing happens under a much broader set of conditions and is more flexible.

After examining the relative costs of operations, it should be clear that memory accesses are the most critical of the performance considerations. NVIDIA GPUs are high-latency, high-bandwidth devices which rely on computation to hide latency from non-cached global memory, and fast local memory is a scant resource. More specifically, the GeForce 8800GTS, a compute capability 1.0 card, only has 8196 registers and 8196 bytes (2048 words) of shared memory per SIMD processor, as shown in Table 3.2. At 64 threads per block (minimum recommended in the CUDA Programming Guide[23]), this amounts to 128 registers per thread with one block on the SIMD processor and 64 registers per thread with two blocks on the SIMD processor. At the limit of eight blocks per SIMD processor, this amounts to only 16 registers per thread and 256 words of shared memory available to each block. At 256 threads per block, this amounts to 32 registers per thread at one block per SIMD processor, and four registers per thread (along with 256 words of shared memory between the threads) at the maximum of 8 blocks per SIMD processor.

For comparison, the MasPar MP-1, a classical SIMD machine from 1990, had 16k bytes of memory per processing element[35], and this machine was decidedly more

"silicon-challenged" than a current GPU. A more modern comparison could be made with the Cell BE (Broadband Engine), which has 256k per SPE[11]. Given the scarcity of local GPU memory compared to similar architectures, along with the aforementioned high-latency, high-bandwidth nature of the device, extra attention must be paid to local memory usage. Expensive re-computation of intermediate values may be relatively cheap compared to caching. Also, prioritizing coalesced memory accesses over full PE utilization can yield better performance.

Chapter 4: Virtualization Constraints

One claim of this thesis is that, although there is a virtualization constraint for using message-passing within the GPU, the constraint is actually harmless if the amount of virtualization chosen is already optimal in terms of its interaction model and per-block execution time. More specifically, optimal virtualization is virtualization such that the performance per unit time is nearly as high as possible, and the interaction model between processes is not limited by the choice of the virtualization. This section first describes the ideas of multi-threading and virtualization inside of GPUs, as well as the memory coherence and interaction models. This section next describes a micro-kernel and presents empirical data which suggests that the optimum amount of threading fully populates the GPU's processing resources some integral number of times, and that virtualization beyond that amount yields no measurable benefits and weakens the interaction model. This section concludes by describing the performance benefits reaped by using an optimum amount of threading, focusing on the implications of having a stronger interaction model and a less-awkward coherence model and how these are required by a message-passing implementation executing within a GPU.

4.1: Background and Definitions

GPUs have evolved a SIMD-based, multi-threaded execution model which relies heavily on virtualization. Multi-threading makes perfect sense in the context of graphics-processing because it allows memory latency to be hidden behind Pixel processing, which increases throughput. More precisely, multi-threading allows data fetches from a slow, non-cached GPU global memory (as discussed in Chapter 2.3) to be hidden behind

useful computations by having a thread de-scheduled while waiting for its memory request to complete. Virtualization also makes perfect sense in the context of graphics-processing because most images processed contain more pixels than available processors. A modern HDTV image, for example, has roughly two million pixels, while a current NVIDIA GPU (the GeForce GTX 280) has 240 processor cores, as shown in Table 3.1. Since each pixel can be processed (more or less) independently, large-scale virtualization of the pixels over the available processors has been (and will remain) an obvious performance win. Large-scale virtualization has also created a nice abstraction between the hardware and the images it processes (since drivers can handle the virtualization), and this abstraction has allowed new generations of GPUs to be developed by merely taking an existing architecture and adding more processing pipelines, tweaking the clock speeds, optimizing the core floating-point units, adding hardware features, etc., rather than inventing a new architecture.

The memory coherence or memory consistency model of global memory in NVIDIA GPUs is slightly awkward. Global memory is the only true shared memory for all the processors on the device, but the only real guarantee about global memory is that specific kinds of transactions are performed atomically. This is not referring to just the atomic operations provided by higher compute-capabilities, but also coalesced memory accesses[23]. Unfortunately, the model provides no rules governing the order of writes, when data written becomes available to other processes, or the fairness of the writer scheduling. Here, fairness refers to threads being guaranteed a chance to write their data, and it is possible that one or more threads will never get a chance to write.

Because of these uncertainties, the interaction model of the processes is relatively weak. A process cannot safely wait on another process, because it is possible that the other process may not be able to run due to a lack of resources. For example, imagine that each process wants to contribute an element to a global array, and then wait until another specific process has contributed before continuing. The interaction model does not allow this, because there could be enough processes created so that all the executing processes wait on contributions from processes queued to wait at the software level, and this would obviously cause a live-lock.

4.2: Performance Analysis of Virtualization

The established method for boosting application performance with GPUs is to write kernels with wide parallelism and allow the GPU to virtualize the computations over the available GPU processing elements. For example, in the NVIDIA CUDA Programming Guide[23] developers are informed that using a large number of thread blocks amortizes the overhead from device memory reads and thread synchronizations better than a small number of thread blocks, and that the number of thread blocks is usually dictated by the size of the data being processed, which can greatly exceed the number of processors in the GPU.

There are established rules which govern the choices of multi-threading and virtualization parameters. CUDA developers are instructed to carefully choose the number of threads per block and/or number of blocks to maximally utilize the device resources[23]. In terms of multi-threading, the CUDA programming guide suggests 64 threads per block (twice the size of a warp) as a minimum, and further suggests that 192 or 256 threads per block is a better number[23]. Obviously, these sizes create some integral number of fully-populated warps. In terms of virtualization, the CUDA Programming Guide states that "It is therefore usually better to allow for two or more blocks to be active on each multiprocessor to allow overlap between blocks that wait and blocks that can run," and that "More thread blocks stream in pipeline fashion through the device and amortize overhead even more. The number of blocks per grid should be at least 100 if one wants it to scale to future devices; 1,000 blocks will scale across several generations."[23]. Because there is no clear message stating "how much is enough," this section presents a micro-kernel which is used to test the benefits of large-scale virtualization.

The NVIDIA CUDA Programming Guide[23] suggests a typical processing pattern of [1)]reading data from global device memory into shared memory, [2)]synchronizing all threads in each block to make sure that all global memory entries have been fetched into shared memory, [3)]processing the data in shared memory, [4)]synchronizing all threads in each block to ensure that all processing results are present in shared memory, and [5)] writing the results back to global device memory. Using the above ideas, a very simple, ideal kernel can be constructed which tests the benefits of large-scale virtualization. The

pseudo-code for this kernel is shown in Figure 4.1.

```
float a,b;

int main()
{
      float ta,tb;
      int i = 10000;
      do {
            ta += tb * 1.11; tb += ta * 1.11;
            ta += tb * 1.11; tb += ta * 1.11;
            ta += tb * 1.11; tb += ta * 1.11;
            ta += tb * 1.11; tb += ta * 1.11;
            ta += tb * 1.11; tb += ta * 1.11;
      } while (--i);

      a = ta;
      b = tb;
}
```

Figure 4.1: Pseudo-code for the micro-benchmark

Note that this kernel is not intended to mimic a complicated or realistic GPU kernel, because realistic kernels introduce extra factors and lack performance transparency. This kernel is arithmetically intensive, providing 10 floating-point multiply-adds in a loop iterating 10,000 times, which provides roughly 400,000 cycles of arithmetic for the two memory reads and two memory writes performed by each thread. The data is aligned in global device memory so that maximum bandwidth can be utilized, and the code requires a minimal number of registers. Under these minimal conditions, the benefits of virtualization are clearly seen.

Two versions of the kernel are tested below with varying block counts (amounts of virtualization): one with `ta` and `tb` in registers, the other with `ta` and `tb` in shared memory. Both kernel versions use 256 threads per block (`BNPROC` is the number of thread blocks). The test GPU is an NVIDIA GeForce 8800 GTS (a compute capability 1.0 card with 12 SMs), and the kernels are executed with varying thread-block counts from 2 blocks to 434 blocks (more than 434 blocks becomes unstable during some tests). After each kernel listing, the average execution time of 10 iterations of the kernel is plotted against the block count. The per-block execution time is then plotted from this data.

Kernel 1, shown in Figure 4.2, performs the operations with the values read into

registers prior to entering the do-while loop and writes the register contents back to global memory before exiting. Both execution time plots are shown following the kernel code, and plots are marked in increments of 24 because this is two times the number of SMs in the test card.

```
#define BNPROC 256
__global__ void
myperf(register volatile float *a, register volatile float *b)
{
  register int IPROC = (blockIdx.x * BNPROC) + threadIdx.x;

  register float a_reg = a[IPROC];
  register float b_reg = b[IPROC];

  __syncthreads();
  int i = 10000;
  do {
    a_reg += b_reg * 1.11; b_reg += a_reg * 1.11;
    a_reg += b_reg * 1.11; b_reg += a_reg * 1.11;
    a_reg += b_reg * 1.11; b_reg += a_reg * 1.11;
    a_reg += b_reg * 1.11; b_reg += a_reg * 1.11;
    a_reg += b_reg * 1.11; b_reg += a_reg * 1.11;
  } while (--i);

  a[IPROC] = a_reg;
  b[IPROC] = b_reg;
  __syncthreads();
}
```

Figure 4.2: The register micro-benchmark

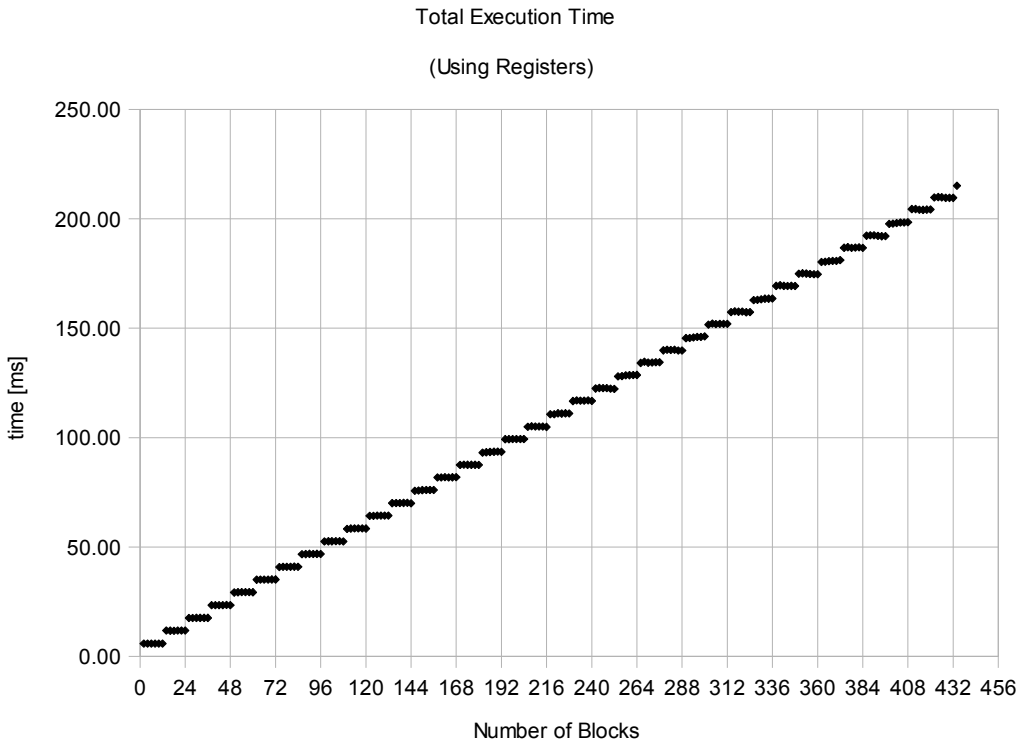

Figure 4.3: Total execution time of the register micro-benchmark

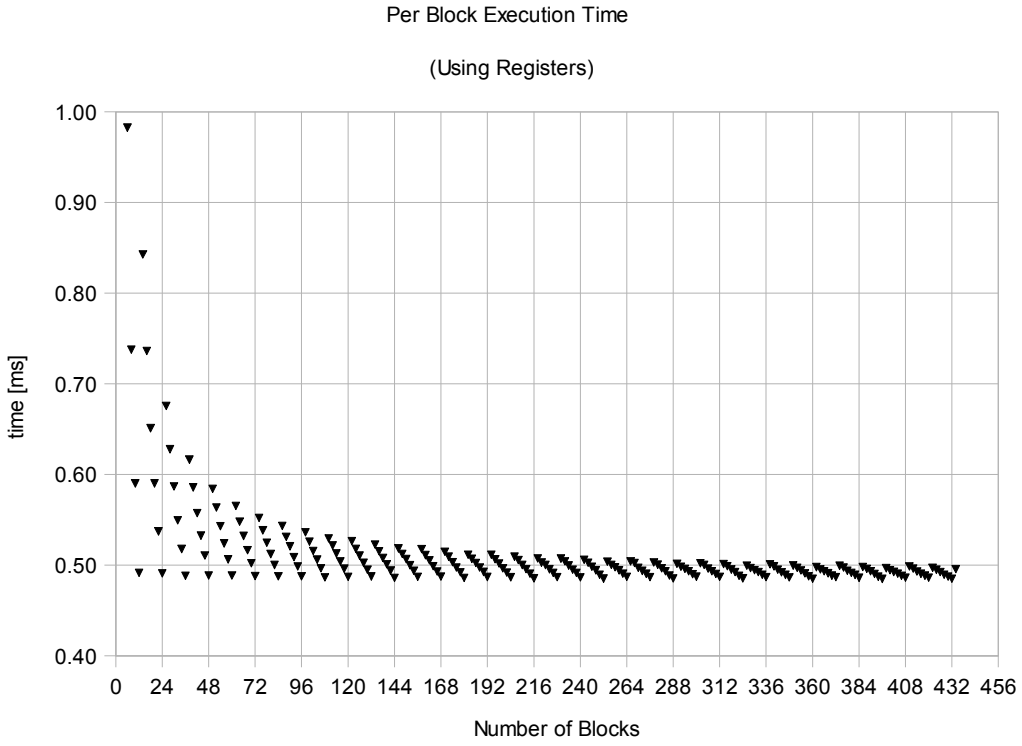

Figure 4.4: Per-block execution time of the register micro-benchmark

Figure 4.3 may be surprising in that execution time is obviously a stair-step function for computationally-intensive code looping in registers. Note that the NVIDIA 8800 used in the test has 12 SMs; not surprisingly every time a multiple of 12 blocks is exceeded the execution time "steps". This is because adding that extra block causes one SM to execute another set of kernels while the other 11 SMs are idle. Adding additional blocks populates the idle SMs, which does not cause any increase in execution time. The important idea is simply that the stair-step characteristic comes from populating the SMs unevenly, and therefore the number of blocks should be chosen based on the number of SMs in a device.

Figure 4.4 shows the execution times normalized to the number of blocks. The actual smallest execution time in this chart occurs at 396 blocks (33 blocks/SM) with a time per block of 0.48499 ms. It should be noted that the time per block at 12 blocks (1 block/SM) of 0.4915 ms is still smaller than the time per block at 390 blocks (0.4934 ms), 388 blocks (0.4959 ms), and 386 blocks (0.4981 ms). Moreover, the percent difference between the time at 12 blocks and the time at 396 blocks is only 1.33%.

From the above data, it can clearly be seen that the near-optimal execution time per block is obtained any time the execution configuration runs a multiple of 12 blocks total, which is the same as running an identical number of blocks on every SM or fully populating the device some integral number of times. However, running a large multiple of 12 yields little additional performance benefit over running a small multiple of 12.

Kernel 2, shown in Figure 4.5, performs the same operations as kernel 1, but reads the global data into shared memory, uses pointers in registers to volatile shared memory inside the loop, and writes the shared-memory values back to global memory upon completion. Both execution time plots are shown following the kernel code, and plots are marked in increments of 24 because this is two times the number of SMs in the test card.

```
#define BNPROC 256
typedef struct {
  float a_s[BNPROC];
  float b_s[BNPROC];
} my_shared_t;

__global__ void
myperf(register volatile float *a, register volatile float *b)
{
  register int IPROC = (blockIdx.x * BNPROC) + threadIdx.x;

  extern __shared__ my_shared_t shared[];
  register volatile float *a_shr = &((*shared).a_s[IPROC]);
  register volatile float *b_shr = &((*shared).b_s[IPROC]);
  *a_shr = a[IPROC];
  *b_shr = b[IPROC];

  __syncthreads();
  int i = 10000;
  do {
    *a_shr += *b_shr * 1.11; *b_shr += *a_shr * 1.11;
    *a_shr += *b_shr * 1.11; *b_shr += *a_shr * 1.11;
    *a_shr += *b_shr * 1.11; *b_shr += *a_shr * 1.11;
    *a_shr += *b_shr * 1.11; *b_shr += *a_shr * 1.11;
    *a_shr += *b_shr * 1.11; *b_shr += *a_shr * 1.11;
  } while (--i);
  __syncthreads();
  a[IPROC] = *a_shr;
  b[IPROC] = *b_shr;
  __syncthreads();
}
```

Figure 4.5: The shared-memory micro-benchmark

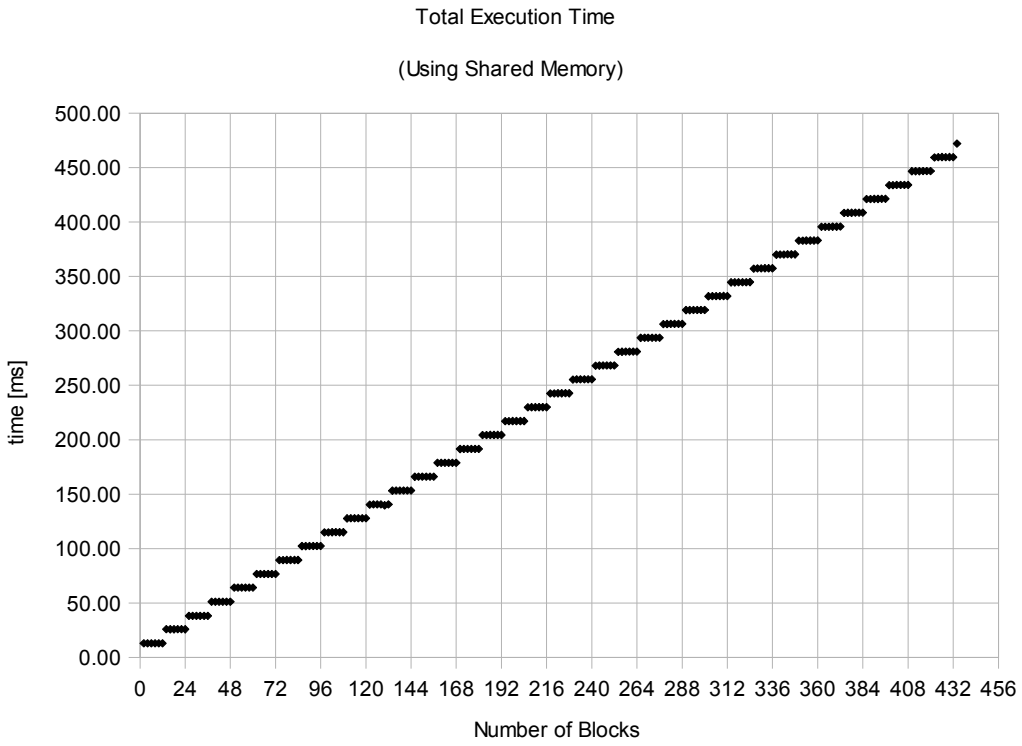

Figure 4.6: Total execution time of the shared-memory micro-benchmark

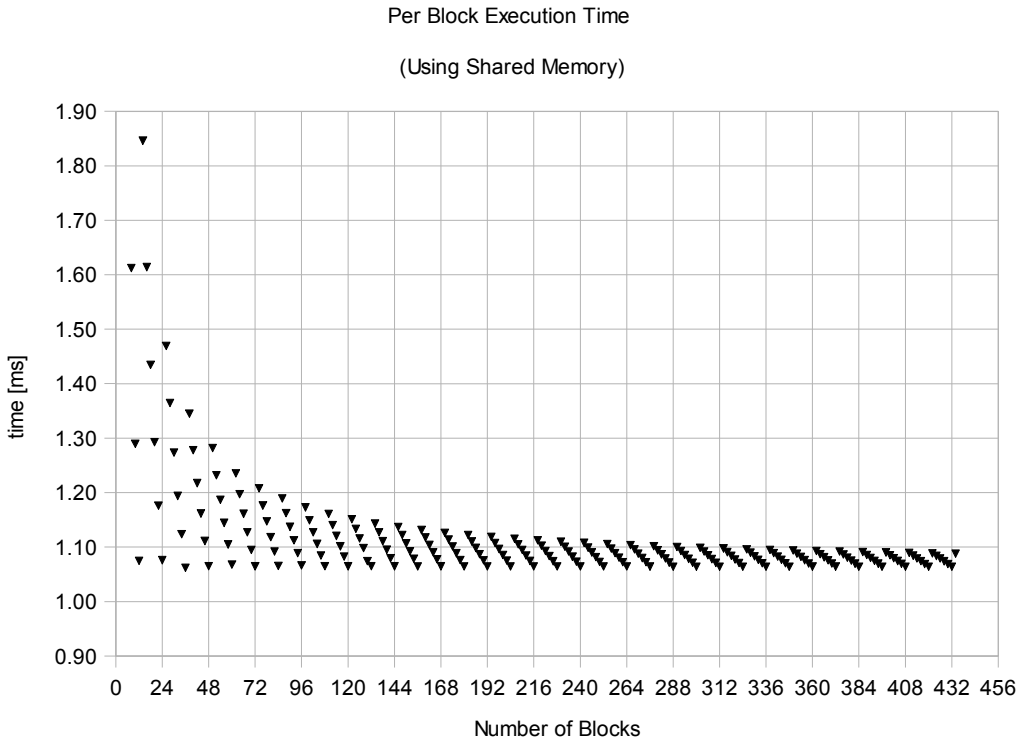

Figure 4.7: Per-block execution time of the shared-memory micro-benchmark

Figure 4.6 shows the same stair-step function for execution time as Figure 4.3. Since conflict-free shared memory accesses are as fast as register accesses in CUDA[23], it is expected that the shared memory behaves in a similar manner to the register file with regards to scheduling behavior. Note that the kernel does not attempt to stride shared memory reads or prevent bank conflicts, since the presence or absence of such conflicts should not change the scheduling behavior of the kernels. While the scheduling unit in an SM may swap blocks to cover global memory latency, the overhead of such a swap would likely swamp a shared-memory conflict completely.

As mentioned before, the NVIDIA 8800 used in the test has 12 SMs; not surprisingly every time a multiple of 12 blocks is exceeded the execution time "steps". This is again because adding that extra block causes one SM to execute another set of kernels while the other 11 SMs are idle. Adding additional blocks populates the idle SMs, which does not cause any increase in execution time. The important idea is again that the stair-step characteristic comes from populating the SMs unevenly, and therefore the number of blocks should be chosen based on the number of SMs in a device.

Figure 4.7 shows the execution time normalized to the number of blocks. The smallest execution time occurs at 36 blocks (3 blocks/SM) with a time per block of 1.06187 ms. This can be compared to the largest execution time at a multiple of 12 blocks, which is 24 blocks at a time per block of 1.075 ms. The percent difference is 1.31%, and again the worst choice of a multiple of 12 (24 blocks at 1.075 ms/block) is better than a naive choice of a large number of blocks (424 blocks at 1.0838 ms/block or 426 blocks at 1.079 ms/block).

Again, the data clearly shows that the near-optimal execution time per block is obtained any time the execution configuration runs a multiple of 12 blocks total, which is the same as fully populating the device some integral number of times. As before, running a large multiple of 12 is not necessary to get nearly-optimal execution time per block, and does not appear to provide much benefit.

At this point, it is clear that running a number of thread blocks which is an integer multiple of the number of SMs is required for near-optimal performance. But at what point does more virtualization result in less performance? In terms of execution time, there appears to be no point where adding another 12 thread blocks results in noticeable

performance loss. In terms of the interaction model, however, there is a very noticeable performance loss as soon as any blocks are queued to wait.

In the introductory material of this section, the interaction model of the CUDA GPU was described as relatively weak, primarily because processes cannot wait on each other due to the uncertainty about what processes are currently executing and the awkward global memory coherence model. If it is known that all thread blocks are executing, however, then blocks can wait on each other by polling, and be assured that a write by one process will eventually be visible by all other processes. This constitutes a much stronger interaction model, and is possible if the virtualization is kept within the limits of what can execute concurrently on a given device. This would be dictated by the number of SMs in the device, the number of active blocks per SM or threads per SM supported by the device, and the per-block resource usage.

While the number of SMs, threads-per-SM constraint, and blocks-per-SM constraint are each device-dependent, the per-thread resource usage is program-dependent. The per-thread resource utilization can be calculated prior to execution by using the built-in `nvcc` compiler option `--ptxas-options=-v` to determine register and shared memory usage, and resource usage can also be partially controlled by using the compiler option `--maxrregcount` *`amount`*, where *`amount`* is the maximum number of registers a GPU function can use[23]. By using this compiler output alongside the information from NVIDIA's `deviceQuery` program provided with the default CUDA SDK installation (the compute capability information and number of SMs in the installed devices), scaling parameters which populate the device without queuing any thread blocks to wait can be automatically predicted for a particular kernel.

In this sense, GPU code could be autonomously optimized for a strong interaction model and nearly-best execution time on new GPU hardware by recompilation or dynamic hardware detection at runtime. This would be similar to ATLAS[36], where the software tests various implementations of linear algebra routines and then uses the results to seed a search for the best implementation on a specific machine. In ATLAS, a large time investment at compile-time pays a large performance dividend at run-time. GPU code can also be automatically tuned at compile-time to use optimal virtualization.

4.3: Optimal Virtualization and the MPI Constraints

Optimal virtualization obviously boasts a performance boost over a poor choice of virtualization (a non-multiple of the number of SMs) in terms of execution time per operation, as shown above. The more important benefit, though, is that the stronger interaction model provided by the virtualization allows communication within a GPU.

Since processes are guaranteed to eventually see values written by other processes, inter-process communication and inter-process synchronization become possible without the need for the traditional kernel stop and restart. Communication can be provided by simple data structures declared `volatile` in global device memory, or by a more advanced mechanism such as message-passing. The benefit of the latter is that it completely hides the remaining shared-memory semantics issues associated with global memory, mainly the non-determinism associated with writes, since there is no guaranteed ordering between processes. Synchronization can be provided by an established single-writer, multiple-reader algorithm, originally proposed and utilized in the SHMAPERS library[37] and adapted to an NVIDIA CUDA GPU within this research group.

There is also a performance impact of the new capabilities: the kernel can continue execution. NVIDIA has stated that kernel launches have low overhead and should be used as global thread synchronization points[23]. The problem is that the hardware overhead may be low, but the software overhead for a kernel stop and restart is not. In the CUDA system, no local processor state is maintained across kernel invocations, so data must be written to global memory at the end of one invocation, and then read from global memory at the beginning of the next invocation[23]. Unpublished experiments conducted within this research group have shown that restoring state from global memory quickly becomes prohibitively expensive, partially due to the lack of computation available to cover the latency from the extra memory reads. In BGSP, the researchers were content with a 4x performance penalty for having to include save and restore state mechanisms, even when not in use[30].

In addition to the cost of restoring state after synchronization, the stop/restart model complicates programming. Currently, if communication is required, then data must either be copied to the host between kernels and altered appropriately before being copied back, or another kernel must perform the communications using the global

memory between invocations. Any code requiring multiple synchronizations must be rewritten into separate kernels, with the host providing the glue logic to invoke the correct kernels as necessary, and synchronizations nested within control structures exacerbate the problem of restoring state and continuing. One simple example of this would be an iterative solver which is looping through a data structure and needs to check against a quality metric at the end of a set of iterations.

Chapter 5: The MPI Implementation

The primary claim of this thesis is that the message-passing programming model can be conceptually cleaner than the currently ubiquitous data-parallel model by hiding rather than dealing with the quirks of current shared-memory environments. This section first provides a brief introduction to MPI (the Message-Passing Interface), which is the standard used in this research to implement the message-passing model on the GPU. It then describes which parts of the standard are actually implemented and what design considerations are involved in the proof-of-concept implementation. Details are also provided on the key point-to-point communication function implementations. A performance comparison to the native NVIDIA CUDA code on a sample algorithm which computes the value of pi in parallel is then presented for the point-to-point communication functions. This section then provides the same details and performance analysis for the key collective communication function implementations. Some of the other MPI functions which were implemented are briefly discussed. This section concludes by describing the key benefits of the message-passing model compared to the data-parallel model, as demonstrated by the proof-of-concept MPI implementation and the example implementations of the sample algorithm.

5.1: Introduction to the Message-Passing Interface

MPI, the Message-Passing Interface, is a parallel-programming standard which provides a message-passing library interface specification[1]. That is, MPI provides a detailed specification of the various functions and data structures which are externally available in a message-passing library implementation which conforms to the standard. The standard does not specify the implementation details under the interface, and so MPI

can be implemented on a variety of hardware architectures. The main benefit of MPI is therefore the ability to easily write portable code utilizing the message-passing model in C, C++, or Fortran.

As seen in the MPI standard[1], MPI specifies several general categories of functionality which contain the various MPI functions. These include point-to-point communication, data-type creation and management, collective or aggregate operations, process group management, communication context management, process topology descriptions, environmental management and environmental query, info object management, process creation and management, one-sided communication, external interfaces management, parallel file I/O, and a profiling interface[1]. Each category of functionality contains both the functions, whose interfaces are defined in C, C++, and Fortran, and any data-types or constants which must be exposed to the user to interact with the functions.

5.2: Design Philosophies and Restrictions

Since the goal of this research is to provide a proof-of-concept MPI implementation rather than a full one, only some functions from some of the categories above are actually implemented. The emphasis is on proving the viability and benefits of the message-passing model, so point-to-point communications and aggregate operations are the main focus of this work. In addition, a handful of data-type creation and management, environmental management and environmental query, communication context management, and one-sided communications functions are implemented.

The functions implemented are chosen because they represent a large portion of what can be done with MPI, and they conceptually cover what is most often done with MPI. The functions are not just the functions which naturally fit a GPU's programming model and architecture; they are a representative set of functions which should demonstrate the benefits of MPI within a GPU in a relatively complete fashion. The functions implemented are also similar to those in other subset implementations such as AFMPI[38], where the goal is to prove the viability and benefits of MPI on unusual hardware.

The following sections describe the implementation in greater detail, but there are a handful of overarching restrictions which mandate design decisions affecting the entire

implementation. Each of these restrictions is described briefly here, and then in greater detail below. The first restriction is the aliasing of the hierarchical groups inside a GPU, which refers to the interdependence of threads, warps, and thread blocks within an NVIDIA CUDA GPU. This restriction affects the choice of what constitutes an MPI process in the implementation. The second restriction is the inability to read from or write to an SM's local shared memory, and this restriction forces all communication to travel through slow global memory. The third restriction is the scarcity of local resources, and this restriction forces some potential optimizations within thread blocks to be ignored and forces a simple name-space for all MPI processes. The fourth restriction is the ease with which the GPU can be accidentally live-locked, and this restriction forces the alteration of some MPI semantics to avoid potential live-locks. The fifth and final restriction is the orientation of hardware to 32-bit operations, and this restriction causes all operations to be naturally word-oriented, and prevents some MPI interfaces involving double-precision floating-point numbers from being implemented. Each of these ideas is described in more detail below.

The first restriction, the interconnection of the hierarchical groups inside a GPU, is simply referring to the fact that thread blocks are aliases for groups of warps, and warps are aliases for groups of threads. A single thread never exists if it does not belong to a warp, and that warp never exists if it does not belong to a thread block. This is different than a cluster of workstations, for example, where a single workstation can exist without being part of a cluster.

The implications of this restriction become clear when deciding what exactly constitutes an MPI process or rank. In DCGN[29], Stuart and Davis discuss the problems with choosing either an entire kernel or single threads as a rank, noting that each mapping can have benefits. In this research, though, the focus is on communication within a GPU, rather than communication across GPUs. Within any GPU, the basic blocks mentioned in chapter 3.2 are present. In the NVIDIA CUDA GPU used here, the obvious choice for what constitutes a rank or process would be a thread, a warp, or a thread block. This research chooses to implement processes as threads, since threads are the smallest units performing work inside a GPU. This has the benefit of making kernel code look relatively normal (rather than always collective across warps or thread blocks), and

keeping the programming model easy to use, as opposed to forcing the user to write vector-oriented code for warps or thread blocks. The obvious penalty for this choice is that all potential SIMD divergence issue are exposed by this choice and must be taken into consideration (this will be illustrated in the test algorithm implementations). The future work for this research includes examining other MPI implementations with processes mapped onto warps and thread blocks. Warps could hide SIMD divergence issues while still providing a low-level, vector-oriented programming model. Thread blocks could hide all scheduling issues when optimally virtualized and provide a MIMD programming model, but at the cost of load-balancing issues and a more complex vector model.

The second restriction involves the limitations on which thread can write to a SIMD processor's local shared memory. Only a thread which is executing in a thread block on a given SIMD processor may read and write to the shared memory allocated to the thread block locally in that SIMD processor. A thread cannot write to shared memory in a thread block executing on another SIMD processor, and cannot write to statically-allocated local shared memory of another thread block executing on the same SIMD processor (a pointer to dynamically-allocated shared memory created by the host can be shared among thread blocks on a SIMD processor, but the operations on it must be synchronized by a global barrier synchronization of all processes, since the `__syncthreads()` command only synchronizes threads within a thread block). This forces communication to move through device global memory (optimizations for threads in the same thread block are not used because of the scarcity of local resources, described in the next section). For send and receive operations, a send may specify a source in shared memory and a receive may specify a destination in a different shared memory, but the message must move through global memory at some point. This also constrains one-sided communication to targets in global memory or in the shared memory of the thread block of the invoking process.

The third restriction, the scarcity of local resources, refers to the relatively small amounts of fast, local resources in current GPUs. On the NVIDIA GPU, these resources include the registers and shared memory, as described in chapter 3.2.

There are two design decisions caused by the restriction: a single-context, flat

process name-space and a lack of optimizations for threads communicating within a thread block. MPI specifies a range of functions for subdividing the collection of all processes into subgroups and for managing unique communication contexts within groups and subgroups. Such functions require that information about the various groups and contexts be available to each process, but this implies that something must be stored for each process to identify its contexts and groups. Given the already-limited amount of local shared memory and registers, and given that the subgroups and communicators are not essential for proving the viability and conceptual cleanliness of the message-passing model, this research chooses not to implement the subgroup and communication context functionality and instead uses a single communicator: `MPI_COMM_WORLD`. The lack of local resources also limits potential optimizations for threads communicating or working within a thread block. As described in the previous section, threads communicating within a thread block could benefit from passing messages through shared memory. However, this optimization could easily consume all the shared-memory resources and leave none for user computation. As an example, if each of 256 threads in a thread block wanted to send 1 single-precision float to the next process, this would require 255*4 = 1020 bytes of shared memory (1/8 the total available if only one thread block is executing on the SIMD processor) for a single buffered send operation.

the ease with which the GPU can be accidentally live-locked, and this restriction forces

The fourth restriction is the the ease with which the GPU can be accidentally live-locked. Because MPI requires some operations to be semantically blocking, there is always the possibility of spin-locking one process while waiting on another which will not be scheduled because of the fairness issues described in chapter 4.1. Note that the choice of a process mapping does not affect this; at optimal virtualization there are still usually more blocks than SMs, and nearly always more threads than physical processors. The mapping of a process to a thread only makes it simpler to hang the hardware via SIMD enable masking used within a warp. Note that the NVIDIA GPU does not support any interrupt mechanism to recover from a lock, the only supported mechanism is a watchdog timer which terminates the kernel invocation after about 10 seconds. When the watchdog timer activates, no local state is preserved and the results of any pending memory writes are undefined.

The live-lock restriction implies that not all blocking operations can be trivially implemented. This research handles this restriction by obeying the semantics of MPI blocking operations without actually blocking where possible. An example of this is `MPI_Send`, which always buffers send data to a system buffer. If buffer space is unavailable, `MPI_Send` returns an error `(MPI_NO_SPACE)` rather than blocking. This also relates back to the choice of a process mapping: a true blocking send and receive pair could never work at the thread granularity due to the enable masking used to implement divergence. Some MPI interfaces, such as `MPI_Recv` and `MPI_Barrier`, must block. As long as the user kernel is written correctly, though, these cannot actually lock the device. The restrictions for using interfaces are described in the following sections alongside the actual interfaces.

The fifth and final restriction is the orientation of hardware to 32-bit operations. Some GPUs still lack support for double-precision, including the compute-capability 1.0 GeForce 8800GTS used in this research. These devices may silently demote double-precision floating-point numbers and computations to single-precision equivalents[23]. Additionally, the cost for utilizing the support tends to be high compared to the native single-precision support for which GPUs became known.

Because of this restriction, this research attempts to utilize 32-bit words wherever possible. All data copy routines operate on 32-bit words, and shorter or longer data-types (including double-precision floating-point) are not supported in computation functions such as reduction. Support for double-precision floating-point could potentially be emulated using pairs of single-precision floating point numbers, as suggested by Dietz[39], but this optimization is beyond the scope of proving the viability and benefits of MPI within a GPU.

Now that the design philosophies and restrictions have been explained, the next sections will describe the important interfaces in detail. There are a total of 28 functions in the proof-of-concept implementation, but only the key ones are explained in detail in their respective sections below. There are 7 point-to-point communication interfaces implemented: `MPI_Send`, `MPI_Recv`, `MPI_Get_count`, `MPI_Iprobe`, `MPI_Probe`, `MPI_Sendrecv`, and `MPI_Sendrecv_replace`. There are 8 collective communication interfaces implemented: `MPI_Barrier`, `MPI_Bcast`,

MPI_Gather, MPI_Scatter, MPI_Allgather, MPI_Alltoall, MPI_Reduce, and MPI_Allreduce. There are 3 data-type management interfaces implemented: MPI_Pack, MPI_Unpack, and MPI_Pack_size. There are 2 communication context management interfaces implemented: MPI_Comm_size and MPI_Comm_rank. There are 5 one-sided communication interfaces implemented: MPI_Win_create, MPI_Win_free, MPI_Put, MPI_Get, and MPI_Win_fence. Finally, there are 3 environmental management and inquiry interfaces implemented: MPI_Get_version, MPI_Init, and MPI_Finalize.

5.3: The Point-to-Point Communication Interfaces

Point-to-point communications are perhaps the quintessential message-passing operations. This section will focus on the send and receive implementations by describing the general point-to-point model, showing pseudo-code for the send and receive operations, and explaining each operation thoroughly. Potential performance improvements are also discussed.

The general model for point-to-point communication inside the GPU is that messages and envelopes are buffered in global memory by the process which sends the message. The buffer is a system data structure (described later in this section), and there is a static limit on the number of buffered messages and the size of each message. The receive operation searches in the buffer of the source it intends to read from, blocking until it finds a matching envelope and message. Once the receive reads the message, it marks the send buffer as read and returns. To better understand these steps, MPI_Send and MPI_Recv are each examined in detail below.

The prototype for MPI_Send is shown in Figure 5.1.

```
/* Send, standard-mode */
__device__ int MPI_Send (
      void* buf,               /* IN */
      int count,               /* IN */
      MPI_Datatype data-type, /* IN */
      int dest,                /* IN */
      int tag,                 /* IN */
      MPI_Comm comm            /* IN */
);
```

Figure 5.1: MPI_Send prototype

The `MPI_Send` function sends `count` items of the type `data-type` from the buffer `*buf` to the process with rank `dest` with an envelope tag value `tag` and the communicator `comm`. The pseudo-code for the entire function is shown in Figure 5.2 (the actual code implementing the function is shown in Appendix A).

```
/* Send, standard-mode */
__device__ int MPI_Send(...)
{
      /* Declare any necessary variables */
      (...)

      /* Check arguments for errors */
      if((err_code = check_args(...))) return(err_code);

      /* Reset received buffers as available and
       *    adjust the total buffered message count
       *    accordingly */
      msg_count = reset_received_buffers();

      /* Ensure that buffer space is still available */
      if (msg_count == MAX_BUFFERED_MESSAGES)
            return(MPI_ERR_NO_SPACE);

      /* Now find the first free buffer
       *    (one marked as available) */
      msg_slot = get_first_available_buffer();

      /* Fill the buffer with the message header */
      set_msg_header(msg_slot,...);

      /* Copy the data from the send buffer
       *    to the the system buffer */
      copy_data(msg_slot,buf,data-type,count);

      /* Increment the current message count */
      set_msg_count(++msg_count);

      /* Serialize the message, which marks it as valid. */
      set_msg_serial(msg_slot, serial_number++);

      /* Return */
      return (MPI_SUCCESS);

}
```

Figure 5.2: Pseudo-code for MPI_Send

The send function is fairly straightforward. First, the necessary temporary variables are declared and initialized. Next, the input arguments are checked for errors. In the case of `MPI_Send`, errors include a destination process which is negative and not `MPI_PROC_NULL`, a tag value which is less than the specified lower-bound `MPI_TAG_LB`, greater than the specified upper-bound `MPI_TAG_UB`, or `MPI_ANY_TAG`, a send type which is not a valid data-type, a send count which is zero,

negative, or would require more storage than `MAX_DATA_PER_MESSAGE*4` bytes, and a communicator which is not `MPI_COMM_WORLD`.

Once error checking is complete, the function needs to update its internal list of message serial numbers. Messages are serialized from 0 upwards, so that two messages sent from one process will arrive at another process in order. The serial numbers are also used to store the special tags `SN_AVAILABLE` and `SN_RECEIVED`, which indicate that a message slot is available for use, or has been received by another process, respectively. The update process consists of resetting all messages marked `SN_RECEIVED` to `SN_AVAILABLE`, so that the send operation can use them again, and decrementing the message count accordingly.

After updating the serial number list, the routine ensures that space is available. If it is, the first message slot marked `SN_AVAILABLE` is located and used for the message. The message header and envelope information is then copied into the slot. This includes the destination, data-type, count, tag, and communicator.

Once the header is written, the actual data must be copied into the system buffer. Although the system buffer is aligned, it is possible for the source buffer to be misaligned. For this reason, the copy operation must get the aligned address of the source, convert the count to bytes, and then perform a different copy routine based on the misalignment of the source address. The aligned-copy routine is a straightforward word-to-word copy. The other copy routines all require that two words from the source buffer be read and then shifted, bit-masked, and pasted together to make the correct aligned data. The routine actually may read more data than specified for code simplicity, but the correct amount of data is stored in the header information.

After copying the data, the process must first increment its message count, then serialize this message with the current serial number and increment the serial number. The ordering of these operations is important, since the receive operation loops based on the message count. Finally, the message and envelope are all buffered and the routine can return successfully.

The prototype for `MPI_Recv` is shown in Figure 5.3.

```
/* Receive */
__device__ int
MPI_Recv
(
      void* buf,              /* OUT */
      int count,              /* IN */
      MPI_Datatype data-type, /* IN */
      int source,             /* IN */
      int tag,                /* IN */
      MPI_Comm comm,          /* IN */
      MPI_Status *status      /* OUT */
);
```

Figure 5.3: MPI_Recv Prototype

The `MPI_Recv` function receives up to `count` items of the type `data-type` into the buffer `*buf` from the process with rank `source` with an envelope tag value `tag` and the communicator `comm`. `MPI_Recv` also returns information about its execution in `*status`. The pseudo-code for the entire function is shown in Figure 5.4 (the actual code implementing the function is shown in Appendix A).

```
/* Recv */
__device__ int MPI_Recv(...)
{
      /* Declare any necessary variables */
      (...)

      /* Check arguments for errors */
      if((err_code = check_args(...))) return(err_code);

      /* This is a blocking receive operation, so
       *    loop until a matching send is found */
      msg_slot = -1;
      while( msg_slot == -1 ) {
            msg_slot = match_envelope(...);
      }

      /* We now have a matching message slot, and
       *    the message will be consumed even if there
       *    are errors, so set the status object. */
      set_status(...);

      /* Check that the data-types match */
      if((err_code = check_types(...))) return(err_code);

      /* Ensure that recv buffer can hold message */
      if( err_code = check_sizes(...))) return(err_code);

      /* Copy the data from the system buffer
       *    to the the recv buffer */
      copy_data(msg_slot,buf,data-type,count);

      /* Mark the message as received */
      set_msg_serial(msg_slot, serial_number++);

      /* Return */
      return (MPI_SUCCESS);
}
```

Figure 5.4: Pseudo-code for MPI_Recv

The recv function is also fairly straightforward. First, the necessary temporary variables are declared and initialized. Next, the input arguments are checked for errors. In the case of `MPI_Recv`, errors include a source process which is negative and not `MPI_PROC_NULL`, a tag value which is less than the specified lower-bound `MPI_TAG_LB` or greater than the specified upper-bound `MPI_TAG_UB`, and not `MPI_ANY_TAG`, a receive type which is not a valid data-type, a send count which is zero, negative, or would require more storage than `MAX_DATA_PER_MESSAGE*4`

bytes, and a communicator which is not MPI_COMM_WORLD.

Once error checking is complete, the function needs to find the message it will receive. To do this, it spins in a loop repeatedly checking the messages in the process specified by source. The actual checking consists of reading the message count from the specified process, and then iterating through the messages until either the maximum number of buffered messages are checked or the number of valid messages read from the source process is the same as the message count read earlier, which would indicate that all valid messages have been checked. Checking a message consists of looking at all messages which are not marked SN_AVAILABLE. A message matches if it is not marked SN_RECEIVED, the tag value and communicator match those specified by the receive, the destination is the rank of the process performing the receive, and the serial number is the lower than the lowest serial number which the receiver has seen so far. Both the serial numbers and the need to check all buffered messages instead of stopping at the first match are dictated by the MPI requirement that messages be non-overtaking[1]. If no match is found, the message count will be re-fetched from the source process and the search will continue.

Upon finding a matching message, the MPI_SOURCE and MPI_TAG fields of the status object are set, since the message will be received at this point even if an error is generated. The count is also set to the count read from the matching message slot, since only that many entries can be received. The data-type is then read from the matching message slot and compared to the receive data-type according to MPI type-matching rules (these essentially state that the types must match perfectly unless one type is MPI_PACKED[1]). If the data-types do not match, the receive call marks the message SN_RECEIVED in the send buffer and returns.

Once the types are matched, the only remaining check is that the count of elements in the send buffer is less than or equal to the count to be received. If the count is greater than the number to be received, the message is marked SN_RECEIVED, but no data is read and the receiver returns with an error code of MPI_ERR_TRUNCATE.

After the checks are complete, the actual data must be copied into the receive buffer. Although the system buffer is aligned, it is possible for the receive buffer to be

misaligned. For this reason, the copy operation must get the aligned address, convert the count to bytes, and then perform a different copy routine based on the misalignment of the receive address. The copies are all the same type as those in `MPI_Send`, but with the exception that the number of bytes written at the end must be exact, and extra data cannot be written. This is not merely a design decision; it is a strict requirement in the MPI standard[1]. The aligned-copy routine is a straightforward word-to-word copy. The other copy routines all require that two words from the system send buffer be read and then shifted, bit-masked, and pasted together to make the correct misaligned data in the receive buffer. After copying the data, the receive process must mark the message serial number as `SN_RECEIVED` in the system send buffer, and can then return successfully.

Now that both `MPI_Send` and `MPI_Recv` have been explained, there are several important concepts and clarifications which must be discussed. These fall into three categories: performance optimizations used in the code, future performance optimizations not implemented yet, and unsupported functionality.

As a performance optimization, this algorithm breaks the strict "owner-writes" rules typical of GPUs by allowing the receiver to write into the serial number array of the system send buffer. NVIDIA CUDA allows arbitrary writers on global memory at the cost of not having cached global memory accesses, while ATI still requires owner-writes global memory access (with the exception of scatter via the "global buffer") but has cached global memory accesses[23][5]. The good news is that some newer models (ie OpenCL[26]) do not have owner-writes restrictions, and there is no problem implementing the algorithm on a system which does, but the bad news is that it requires additional space on the receive process. In particular, the receiver would record in its global space that it had read a message with some envelope and serial number from some sender, and each sending process would have to check all its targets during sends and at the next global synchronization to see what messages had been read (Global synchronization is required for the receiving process to stop indicating it received a certain message, since this ends the cycle of "process 1 saw that process 2 saw that process 1 saw that process 2 saw that …" etc.). Since receiving processes would need to keep records of all messages that had been received until a global synchronization, the buffer holding the records in global memory could in theory be very large for each

process.

The next performance optimizations are both related to the arrangement of the message data in the system send buffer. All the handles which access the system send buffer are written in macros, which hides the nastiness of accessing arrays of structures of special data-types and arrays. More importantly, this allows memory layouts to be tweaked to enable better coalescing and memory access patterns without major recoding.

Originally, the global message buffer was laid out as an array of process message structures. Each process message structure was then laid out as an array of individual message structures and some header information, including the current serial number, the message count, and the serial number array. The individual message structures, which were the lowest level, contained the header information for each message and an array of data which was used to store the message contents.

After seeing that the aggregate communications and one-sided communications could share the system message buffer for storage space, the process message structures were re-written so that all the data would be at the process level. The buffers are still statically-sized and have fixed limits for send and receive operations, but this allowed aggregate communications hijacking the buffer to store as much data as all the message buffers in a process could hold, and to store it in a simple manor without jumping between multiple buffers.

There are other optimizations which could be made. First and foremost, all the message buffers for all processes should be joined at the top level as a single array in the system message buffer, as this (combined with an appropriate skew) could potentially allow send operations to coalesce. The skew is required so that the threads of a half warp will access consecutive addresses when accessing the same element in their respective system buffers.

Other potential optimizations could be provided by allowing the user to promise that only aligned addresses are passed, thereby reducing the computation and memory accesses needed to perform a copy; or by allowing shared-memory usage for point-to-point messaging within a thread block, which would drastically reduce the costs of sending certain messages at the aforementioned expense of using a lot of a scarce local resource. It should also be noted that, since the MPI implementation does not allow user-

defined communicators or data-types, all data-types and communicators could be eliminated at compile-time. The macros for determining the size of a data-type are already evaluated at compile-time, but more code could likely be removed.

Finally, there are two capabilities missing in the current `MPI_Send` and `MPI_Recv` implementations: receive from any source (`MPI_ANY_SOURCE`) and receive with any tag (`MPI_ANY_TAG`). The `MPI_ANY_TAG` support is not implemented yet simply because the research has focused on optimizations with more potential benefit, although it can be implemented by merely modifying the envelope-checking routine in the `MPI_Recv` code. Support for `MPI_ANY_SOURCE` has not been implemented because it implies a search in global memory. Instead of merely searching the specified source for a matching message, the receiving process would have to search through as many processes as necessary to find a matching message.

The other supported point-to-point communication interfaces are basically portions or combinations of the `MPI_Send` and `MPI_Recv` functions. `MPI_Iprobe` and `MPI_Probe` are merely non-blocking and blocking envelope-matching routines from `MPI_Recv`, which set a status object to the values of what would have been received if `MPI_Recv` was called with the same parameters. `MPI_Sendrecv`, and `MPI_Sendrecv_replace` are simply wrappers which call `MPI_Send` and then `MPI_Recv`. Finally, `MPI_Get_count` is just a routine which reads a system-defined (as opposed to MPI-standard defined) field in the status object returned by `MPI_Recv` and returns the number of elements which were actually received.

5.4: Point-to-Point Communication Performance

This section will show the performance of the MPI implementation compared to the performance of native CUDA code, with both executing a parallel algorithm which estimates PI. The code for the basic sequential algorithm is shown in Figure 5.5.

```
main(int argc, char **argv)
{
  register double width, sum;
  register int intervals, i;

  /* get the number of intervals */
  intervals = atoi(argv[1]);
  width = 1.0 / intervals;

  /* do the computation */
  sum = 0;
  for (i=0; i<intervals; ++i) {
    register double x = (i + 0.5) * width;
    sum += 4.0 / (1.0 + x * x);
  }
  sum *= width;

  printf("Estimation of PI is %f\n", sum);

  return(0);
}
```

Figure 5.5: Algorithm for sequential estimation of PI

The algorithm computes the value of pi by summing the area under x^2, and was used in the Linux Documentation Project Parallel Processing HOWTO[40] (and originally shown by Quinn[41]) to demonstrate how various programming models differ. If the intervals are distributed among processes, the algorithm instantly becomes a parallel computation, with several possible implementations for transferring and combining the partial sums from each process. This is not a benchmark algorithm which flatters MPI, since it does not contain enough work to show speedup on most parallel machines; this is an algorithm which has been shown to be useful for comparing and contrasting programming environments.

For the point-to-point communication performance tests, the implementations will use a send and receive mechanism to have all processes aggregate their partial sums on process 0. Process 0 will then complete the summation and return the result. Note that this is not a good approach (a reduction would be more appropriate), but the algorithm provides a basis for comparison between native CUDA with and without the MPI implementation. Figure 5.6 shows the kernel for the algorithm implemented in CUDA using MPI, while Figure 5.7 and Figure 5.8 show the kernels for the algorithm

implemented in CUDA without MPI.

```
/* MPI Implementation using point-to-point communication */
__global__ void
PI__point_to_point( register volatile int *interval_p,
                    register volatile float *sum_p,
                    register volatile int *error_p)
{
      register float width, sum;
      __shared__ float lsum[NUM_THREADS_PER_BLOCK];
      register int intervals, i;
      int nproc, iproc, ib;
      MPI_Status status;

      if (MPI_Init((void *) 0) != MPI_SUCCESS) {
            error_p[0] = 1; return;
      }
      MPI_Comm_size(MPI_COMM_WORLD, &nproc);
      MPI_Comm_rank(MPI_COMM_WORLD, &iproc);

      intervals = *interval_p; width = 1.0 / intervals;
      ib = iproc % NUM_THREADS_PER_BLOCK; lsum[ib] = 0;
      for (i=iproc; i<intervals; i+=nproc) {
            register float x = (i + 0.5) * width;
            lsum[ib] += 4.0 / (1.0 + x * x);
      }
      lsum[ib] *= width;

      if (iproc != 0) {
            MPI_Send(&(lsum[ib]), 1, MPI_FLOAT, 0,
                        0, MPI_COMM_WORLD);
      }
      if(iproc == 0) {
            sum = lsum[ib];
            for (i=1; i<nproc; ++i) {
                  MPI_Recv(&(lsum[ib]), 1, MPI_FLOAT, i,
                              0, MPI_COMM_WORLD, &status);
                  sum += lsum[ib];
            }
            sum_p[0] = sum;
      }

      MPI_Finalize(); error_p[iproc] = 0;
      return;
}
```

Figure 5.6: Point-to-point MPI implementation of the test algorithm

```
/* Native CUDA using point-to-point communication */
/* Kernel 1 of 2 */
__global__ void
PI__point_to_point_1(register volatile int *interval_p,
 register volatile float *lsum_p)
{
      register float width;
      __shared__ float lsum[NUM_THREADS_PER_BLOCK];
      register int intervals, i;
      int nproc, iproc, ib;

      nproc = blockDim.x * gridDim.x;
      iproc = (blockIdx.x * blockDim.x) + threadIdx.x;

      intervals = *interval_p; width = 1.0 / intervals;
      ib = iproc % NUM_THREADS_PER_BLOCK; lsum[ib] = 0;
      for (i=iproc; i<intervals; i+=nproc) {
            register float x = (i + 0.5) * width;
            lsum[ib] += 4.0 / (1.0 + x * x);
      }
      lsum[ib] *= width;

      if (iproc != 0)
      {
            /* Send the partial sum to processor 0 */
            lsum_p[iproc] = lsum[ib];
      }

      return;
}
```

Figure 5.7: Point-to-point CUDA-only implementation of the test algorithm, part 1

```
/* Native CUDA using point-to-point communication */
/* Kernel 2 of 2 */
__global__ void
PI__point_to_point_2(register volatile float *sum_p,
                  register volatile int *error_p,
                  register volatile float *lsum_p)
{
      register int i;
      int iproc,nproc, ib;
      __shared__ float sum;

      nproc = blockDim.x * gridDim.x;
      iproc = (blockIdx.x * blockDim.x) + threadIdx.x;
      ib = iproc % NUM_THREADS_PER_BLOCK;

      if(iproc == 0) {
            sum = lsum_p[iproc];
            for (i=1; i<nproc; ++i) {
                  sum += lsum_p[i];
            }
            sum_p[0] = sum;
      }

      error_p[iproc] = 0;
      return;
}
```

Figure 5.8: Point-to-point CUDA-only implementation of the test algorithm, part 2

There is little to explain regarding the above codes. The MPI implementation uses standard MPI functions to initialize, get the process rank and number of processes, have every process except process 0 send to process 0, and then have process 0 receive all the partial sums. The computations are nearly identical to those in the original algorithm, with the exception that they are performed in a shared-memory block. The native CUDA code uses CUDA built-ins to identify the process and number of processes, and then uses an intermediate memory location to pass out the partial sums to global memory for the next kernel invocation. Again, the computation is identical to the original but performed in shared memory. The second CUDA kernel recalculates the process parameters and then has process 0 read the global memory, summing as it goes. When finished, process 0 records the sum in global memory and all processes return.

To test the execution-time performance of these kernels, each was executed three times with the number of intervals per process varied from 2 to 128 in steps of 2. The

resulting average execution times are shown in Figure 5.9.

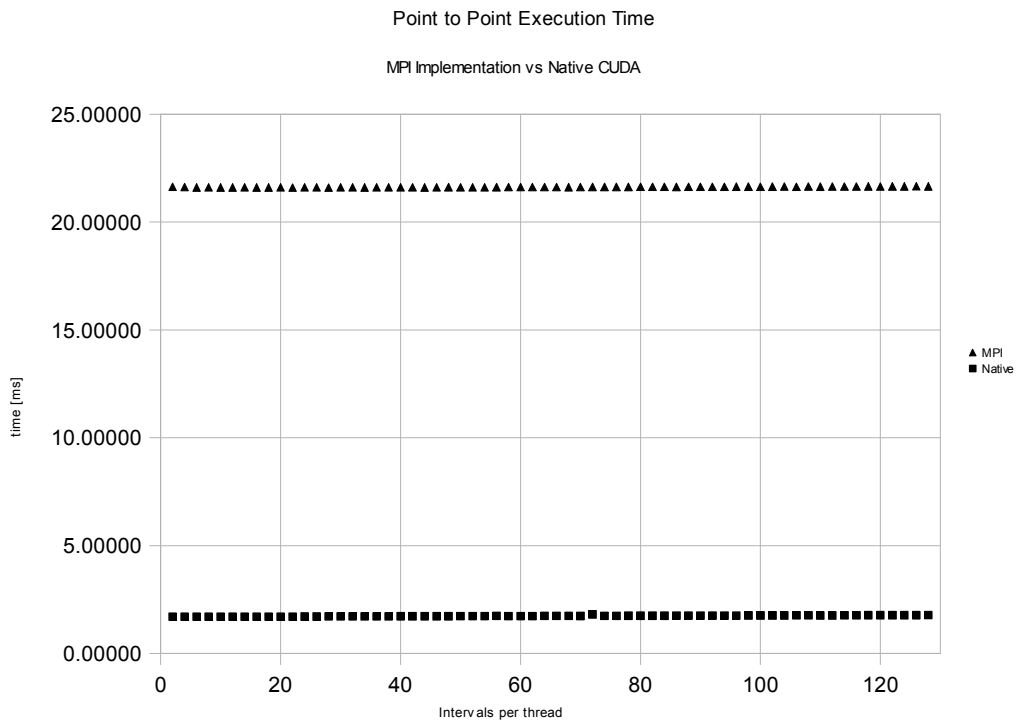

Figure 5.9: Execution time of both implementations of the test algorithm

The above chart shows the execution time for the MPI implementation and the native CUDA implementation. The average execution time of the MPI kernel is 21.62ms, while the average execution time of the native CUDA kernel is 1.74ms. The difference in execution time between 128 intervals per process and 2 intervals for process is 0.014ms for the MPI implementation and 0.079ms for the native CUDA implementation.

The performance data above indicates that there is roughly a 12.5x execution time performance penalty for using the MPI kernel over the native kernels with a point-to-point communication implementation of the PI-estimation algorithm. Examining the execution times can easily reveal the reason. Note that the execution time does not change noticeably when the number of intervals change. This suggests that computation does not factor into the execution time, meaning that memory latency must be the dominating portion of it. While the native CUDA routine only performs one global memory write per process and then a sequential set of as many global reads as there are processes to move the partial sums, it does so by virtue of having the barrier-

synchronization mechanism provided by the first kernel stopping and the second starting. The MPI kernel, on the other hand, does not use any barrier synchronization. Instead, process 0 searches for and fetches messages in global memory. Each send operation, although happening in parallel with others, involves a series of global memory reads and writes for bookkeeping, finding a message slot, and writing the header information, in addition to writing the actual data. Since only one data element is written per process, there is no amortizing of the latencies for the extra reads and writes over multiple data elements. Similarly, the receive operations involve envelope matching and header copying in addition to the actual message copying.

While the message-passing implementation has worse performance in terms of execution time, it certainly has better performance in terms of simplicity and a standardized, familiar interface. The message-passing implementation also hides the shared-memory issues, transparently allowing messages to pass from shared memory in one thread block to shared memory in another thread block.

5.5: The Collective Communication Interfaces

While point-to-point communications involve communication between individual processes, collective communications involve communication within groups of processes. This section will focus on the barrier synchronization and reduction implementations by describing the general collective communication model, showing pseudo-code for the operations, and explaining each operation thoroughly. Potential performance improvements will also be discussed.

The general model for collective communication inside the GPU is a two-step process. First, depending on the operation, one or all processes (the sending processes) will perform some local operations to accomplish half the operation. In a broadcast operation, for instance, the root process would participate by copying the broadcast data to a buffer in global memory, thereby making it accessible to all other processes. In a gather operation, this would consist of all processes writing their contributions to a global space accessible to root. After the first phase of the collective is completed, a barrier synchronization is required.

The second step is basically the same as the first, except it involves the receiving group of processes. In this step, the processes will complete the operation on the now-

visible data from the first group of processes. In a broadcast operation, every process reads the broadcast data at once. In a gather operation, the root process reads the data from the global memory into its result buffer. After this part of the collective, another barrier synchronization is required.

There are some aggregate operations which do not conform to the two-step general model presented above. Personalized all-to-all communications, for example, require O(n) passes through the above steps in a GPU. Barrier synchronization also does not require two phases, as it is more basic than gather or scatter operations. To better understand the collective communications, `MPI_Barrier` and `MPI_Reduce` will each be examined in detail below.

The prototype for MPI_Barrier is shown in Figure 5.10.

```
/* Barrier synchronization */
__device__ int
MPI_Barrier
(
      MPI_Comm comm           /* IN */
);
```

Figure 5.10: MPI_Barrier prototype

The `MPI_Barrier` function performs a barrier synchronization of all the processes in the communicator `comm`. The pseudo-code for the entire function is shown in Figure 5.11 (the actual code implementing the function is shown in Appendix A).

```
/* Barrier synchronization */
/* (Uses a single-writer, multiple-reader mechanism in
 *    global memory) */
__device__ int MPI_Barrier(...)
{
      /* Declare any necessary variables */
      (...)

      /* Check arguments for errors */
      err_code = check_args(...);

      /* Sync here to ensure that everybody in this
       *     block is at the barrier */
      __syncthreads();

      /* The first thread in the block represents the
       *    whole block in the synchronization */
      if (thread_id_in_block == 0)
      {
            /* Increment this block's barrier number */
            bar_nums[my_block]++;

            /* Now, starting at the next block, scan the
             *    barrier numbers until either all
             *    other blocks have arrived or one
             *    block has passed */
            do
            {
                  done = scan_numbers(bar_nums, my_block);
            } while (!done);

            /* The barrier is completed. Increment again
             *    to inform anybody still scanning */
            bar_nums[my_block]++;
      }

      /* All threads wait here for the first thread */
      __syncthreads();

      /* Return */
      return(err_code);
}
```

Figure 5.11: Pseudo-code for MPI_Barrier

The barrier function is essentially a single-writer, multiple-reader mechanism (as mentioned before, the algorithm was first published by Dietz[37], and later adapted to NVIDIA CUDA inside this research group). First, the necessary temporary variables are declared and initialized and the input arguments are checked for errors. In the case of `MPI_Barrier`, the only potential error is using an unsupported communicator (the only

supported communicator is `MPI_COMM_WORLD`). Note that an erroneous function call does not return until it has participated in a barrier, because doing so results in a system hang while the other synchronizing processes wait for it.

Once error checking is complete, the function first obtains a pointer to the array of barrier numbers in global memory (this array is normally initialized to 0 by either `MPI_Init` or the host before a kernel invocation). Next, a `__syncthreads()` command is issued to synchronize the threads in each block, ensuring that all processes are at the barrier. Once this synchronization completes, only the first thread in each block participates in the actual barrier synchronization.

The thread first allocates a register to store the barrier numbers as they are read, a register to store the current barrier number for this block, and a read barrier numbers and a register to hold the starting location for this block's scan (which will be the next block). Once these registers are initialized, the thread increments its barrier number in the global array to indicate that it has arrived at the barrier. It then begins scanning the next block's barrier entry in global memory, waiting for it to be greater than or equal to the scanning thread's current barrier number. If all blocks begin by scanning at process 0 instead, the performance is only slightly degraded. This likely indicates that high latency, rather than a lack of bandwidth, limits the speed of the algorithm.

Once the barrier number being scanned increases, the scanning thread first increments its scan position, looping if necessary, and then checks to see if the barrier is finished. The barrier is finished when either the thread is scanning its own entry in the barrier number array, meaning it made the whole loop through the barrier number entries, or the thread has scanned an entry which had a barrier number greater than the current value for the scanning thread, meaning that the thread being scanned knows that every other process has finished the barrier and is continuing. In either case, the thread exits the loop and increments its own barrier number again to indicate that the barrier synchronization is finished to any processes which are still scanning.

Finally, all processes enter another `__syncthreads()`, which ensures that no process can schedule code while process 0 is still participating in the barrier. Once the `__syncthreads()` command completes, the processes return the error code, which was `MPI_SUCCESS` unless the communicator provided was invalid, in which case it was

MPI_ERR_COMM.

Before the prototype for MPI_Reduce is shown, some explanation is needed. Because of the length of reduction functions, it would be inconvenient to implement each reduction function as a copy with only a few symbols and/or data-types changed. For this reason, the research implements the reduction function as a macro which is called repeatedly to create the various reductions. The macro to declare the prototype, as well as an invocation of it, are shown in Figure 5.12. To simplify discussion, this thesis will use the example of MPI_SUM reduction of MPI_FLOAT data-types. The prototype for this particular reduction function is shown in Figure 5.13.

```
/* This macro creates a reduction function prototype. */
#define CREATE_PROTOTYPE_MPI_REDUCE_4_BYTE_OPS(op,dtype) \
__device__ int \
MPI_Reduce_##dtype##_##op \
( \
      void* sendbuf,    /* IN */ \
      void* recvbuf,    /* OUT */ \
      int count,        /* IN */ \
      int root,         /* IN */ \
      MPI_Comm comm     /* IN */ \
);

/* Declaration of a MPI_SUM reduction of MPI_FLOAT types */
CREATE_PROTOTYPE_MPI_REDUCE_4_BYTE_OPS(MPI_SUM,MPI_FLOAT)
```

Figure 5.12: MPI_Reduce prototype macro

```
/* The created reduction function prototype. */
__device__ int
MPI_Reduce_MPI_FLOAT_MPI_SUM
(
      void* sendbuf,    /* IN */
      void* recvbuf,    /* OUT */
      int count,        /* IN */
      int root,         /* IN */
      MPI_Comm comm     /* IN */
);
```

Figure 5.13: MPI_Reduce prototype

The MPI_Reduce function performs a reduction operation specified by op and datatype on count elements of each process. The elements are located in sendbuf,

and the reduction operation involves all of the processes in the communicator `comm`. The process specified in `root` receives the final reduction result in `recvbuf`. The macros which cause the correct function variant to be invoked are shown in Figure 5.14, while the pseudo-code for the entire `MPI_FLOAT` , `MPI_SUM` variant is shown in Figure 5.15 (the actual code implementing the function is shown in Appendix A).

```
/* This macro changes the MPI_Reduce() function call to a
 *    particular (data-type,op) variant at compile-time. */
#define MPI_Reduce(sbuf,rbuf,count,dtype,op,root,comm) \
MPI_Reduce_##dtype##_##op(sbuf,rbuf,count,root,comm)
```

Figure 5.14: MPI_Reduce translation macro

```
/* Reduction (MPI_SUM for MPI_FLOAT data-types)*/
__device__ int MPI_Reduce_MPI_FLOAT_MPI_SUM(...)
{
      /* Declare any necessary variables */
      (...)

      /* Check arguments for errors */
      err_code = check_args(...);

      /* For each element, have all processes read that
       *    element into shared mem from their sendbuf
       *    and reduce the values in shared mem, then have
       *    the first process in each block write the
       *    value out into global memory. */
      for(i=0 ; i<count ; ++i) {
            reduce_data_tree();
            if(my_id==0) global[my_block] = my_data;
      }

      /* Use a barrier synchronization to ensure that all
       *    values are written in global memory */
      MPI_Barrier(...);

      /* Do a linear reduction using one process per
       *    element, since there are not many blocks. */
      if(my_id < count) {
            fetch_global_data(...);
            reduce_data_linear();
      }

      /* Synchronize root and read the data */
      __syncthreads();
      if(root == MPI_ROOT) {
            copy_data(msg_slot,buf,data-type,count);
      }

      /* Now, synchronize again to let root finish */
      MPI_Barrier(...);

      /* Return */
      return(err_code); \
}
```

Figure 5.15: Pseudo-code for MPI_Reduce

The reduction function may be the most complicated function in the entire implementation, so this section walks through it step by step. Reduction also has some special requirements when compared to other functions in the implementation, and these are mentioned as they arise. It is important to note that, like all aggregate communications, reduction hijacks the system message buffer, so there can be no pending

point-to-point messages waiting or the results of such communications are undefined (usually corrupted message data). Also, MPI_Reduce uses shared memory to hold one reduction element for each process, so the function behaves in an undefined fashion if the shared memory is not available (behavior could include failure of the kernel invocation or corruption of existing data in local shared memory).

First, the necessary temporary variables are declared and initialized and the input arguments are checked for errors. In the case of MPI_Reduce, errors may include a count which is negative, 0, or greater than the max amount of data which a process' message buffer can hold (MAX_DATA_PER_MESSAGE * MAX_BUFFERED_MESSAGES), a communicator which is unsupported, or a root process which is negative or greater than the total number of processes and not MPI_ROOT. Because count is structured as a logarithmic reduction in shared memory, there is an additional restriction that the input buffer on each process be aligned on a word boundary. This is needed to use the smallest amount of shared-memory possible and simplify the reduction code (although it could be removed as a future optimization, as discussed below). Note that, like MPI_Barrier, errors do not cause the function to return; they cause it to participate with an identity entry. In the case above, with a MPI_FLOAT data-type and MPI_SUM operation, processes with errors will contribute the value 0.0f.

Once the error checking is complete, the first reduction begins. With aggregate operations which are focused solely on communicating data, such as MPI_Bcast or MPI_Gather, the next step would be to copy the data to global memory. In MPI_Reduce, this incurs a large performance drop, and the better solution is to copy the values directly to shared memory in each block. There is a large issue with doing so: a lack of local shared memory. Because there is so little shared memory, copying all the reduction vectors from each processor is not feasible. It is feasible to reduce one element from each vector at a time, though, and this code does exactly that. If the input vectors are allowed to be misaligned, though, the storage cost instantly triples. In addition to one array with the length of the number of processes (to hold the actual reduction data), two more identically-sized arrays are required to hold the elements used to paste together the data in an aligned way. To deal with the above costs, the function requires that input data

be aligned on word boundaries.

The structure of the reduction is straight-forward. For each element in the reduction vectors, the processes load their individual elements at that index into the shared memory they control. Then, after a `__syncthreads()`, a simple reduction is applied which halves the number of active processes, keeping the lower half active (this prevents divergent warps), until only one process (process 0 in the block) is active. Process 0 then writes out the reduced value into its message buffer as an element in message zero (since the messages are linked into a single pool, all elements can be written by merely incrementing the position, rather than requiring distribution between message buffers). This process repeats until all elements in the vector have been reduced and written to global memory.

After the block reductions are completed, a barrier synchronization is used to allow all blocks to write their reduced values. Once this barrier completes, every block can then fetch the values (the overlapping reads turn into broadcasts) and reduce to a final value. This avoids having to identify which block has the root process.

The second reduction is a linear reduction which reduces the vectors of elements, and only processes which have an ID inside the block less than the number of elements can participate. In this reduction, each process first fetches a value from the global message buffer of process 0, and then sums that element with the corresponding elements from the message buffer of the processes at the beginning of each block. The result is a vector in shared memory which contains the final reduction values.

At this point, a `__syncthreads()` is issued to ensure that the root process can start reading the shared-memory values, and then an aligned-to-misaligned copy routine (identical to that in `MPI_Recv`) is used to copy the vector from shared memory over to the root process' receive buffer. Only the root process is enabled for the copy routine, so all processes enter a barrier synchronization after the routine to ensure that the root process has finished before any other processes return. After the barrier synchronization, each process can return its error code, which was `MPI_SUCCESS` unless an error occurred and changed the value.

Now that both `MPI_Barrier` and `MPI_Reduce` have been explained, there are several important concepts and clarifications which must be discussed. These fall into

three categories: performance optimizations used in the code, future performance optimizations not implemented yet, and unsupported functionality.

Beyond the optimizations discussed in the description of `MPI_Reduce`, the most important is that the code does not require a kernel stop and restart, which simplifies it considerably. There is no way to jump to a specific instruction when starting a kernel, so stopping and restarting involves completely restructuring code. Chapter 5.8 will compare the two approaches in more detail.

The other important optimization used in `MPI_Reduce` is the compile-time selection of code. This is a benefit in terms of execution speed, because it eliminates conditionals and allows the compiler to fold out many constants. It is also a benefit in terms of code size. There is limited code space (~2 million ptx instructions) for any kernel[23]. If the MPI implementation will be used for more complex algorithms, then code size may become an issue. Therefore, compile-time selection of code is an optimization which can and should be used in other functions.

Beyond `MPI_Reduce` and `MPI_Barrier`, there is one other optimization worth mentioning, and that is the implementation of sequential writes as parallel reads where possible. In `MPI_Bcast`, the broadcast is performed by the root process writing to its message buffer, and then all processes reading from the message buffer in parallel. This optimization, however, is limited by the inability to read local shared memory remotely. `MPI_Scatter`, for instance, only has one implementation. It makes no difference whether the root process writes the scatter data into its own global message buffer or into the global message buffers of other processes, since the number of writes will still be the same, and the non-root processes cannot read the root shared memory. `MPI_Gather` similarly cannot be implemented by having all processes write to the root process' receive buffer in parallel, since the receive buffer is likely located in shared memory. Instead, the algorithm must be implemented with sequential reads.

There are some potential performance improvements which could still be implemented in the collective communication functions. Obviously, allowing the user to specify aligned inputs can save code space and branch overhead. There are also improved reduction schemes for `MPI_Reduce`, and the second reduction can likely be converted to a tree if a reasonable method of storing the final reduced vector can be

obtained (without storing the vector, writing to the root process' receive buffer, which is an aligned-to-misaligned write, would be required during each iteration). The reductions in `MPI_Reduce` can be rewritten to take non-power-of-two inputs and to reduce more inputs than processes in a block, and both of these would also be helpful optimizations. Also, any collective could benefit by using shared-memory to have the root process send or receive from processes in the same block.

The only issue with some of the above optimizations involves MPI function calls. Imagine, for example, that an improved reduction scheme is used which starts by distributing the elements among the processes so that the first warp reduces the first element, the second reduces the second element, and so on. This allows the first several reduction passes to fully utilize the processors by adding multiple elements in each thread. The issue is locating the elements in other processes. In the `MPI_Reduce` function call, a process specifies its data using a pointer. There is no requirement that the pointers be to consecutive addresses in memory, so simple arithmetic will not suffice for a process to find a data element which it is not passing into the reduction. Instead, something like a list of pointers would likely be required, which implies additional storage. Moreover, the `nvcc` compiler can easily lose track of whether a pointer is addressing global or shared memory, and its action upon doing so is to require that the pointer address global memory only. This means that some manipulation will be required to allow any process to read shared memory to which another process passed a pointer.

There are two capabilities missing in the current collective communications: `MPI_IN_PLACE` support and `MPI_Reduce` support for data-types which are not word-length. `MPI_IN_PLACE` support is left as an optimization for future research, since it appears to have less potential benefit than other work. `MPI_Reduce` support for data-types which are not word-length is important, and the support is already effectively there for some operations on shorter data-types (such as the bitwise AND `MPI_BAND` and the bitwise OR `MPI_BOR`) using SWAR[42] techniques in the 4-byte code. The support for other data-types, however, has not yet been implemented. This includes support for `MPI_DOUBLE`, as mentioned in chapter 5.2, although the future work will provide it via pairs of single-precision floating-point values.

The other supported collective communication interfaces are mostly data-transfer routines utilizing the aligned-to-misaligned and misaligned-to-aligned copy routines along with `MPI_Barrier`. Of the 8 interfaces implemented, the broadcast operation `MPI_Bcast`, the gather operation `MPI_Gather`, and the scatter operation `MPI_Scatter` fall into this category. `MPI_Allgather` is an extension of `MPI_Gather` which returns the results to all processes, and `MPI_Alltoall` is merely a sequence of `MPI_Gather` calls, one executed from each process. Finally, `MPI_Allreduce`. Is merely an extension of `MPI_Reduce` in which all processes perform the final copy to the receive buffer.

5.6: Collective Communication Performance

This section will again show the performance of the MPI implementation compared to the performance of native CUDA code with both executing a parallel algorithm which estimates PI. This time, however, the algorithm will be implemented as a collective reduction operation. The code for the basic sequential algorithm is shown again in Figure 5.16 for reference.

```
main(int argc, char **argv)
{
  register double width, sum;
  register int intervals, i;

  /* get the number of intervals */
  intervals = atoi(argv[1]);
  width = 1.0 / intervals;

  /* do the computation */
  sum = 0;
  for (i=0; i<intervals; ++i) {
    register double x = (i + 0.5) * width;
    sum += 4.0 / (1.0 + x * x);
  }
  sum *= width;

  printf("Estimation of PI is %f\n", sum);

  return(0);
}
```

Figure 5.16: Algorithm for sequential estimation of PI

As mentioned before, the algorithm computes the value of pi by summing the area under x^2, and was used in the Linux Documentation Project Parallel Processing HOWTO[40] (and originally shown by Quinn[41]) to demonstrate how various programming models differ. If the intervals are distributed among processes, the algorithm instantly becomes a parallel computation, with several possible implementations for transferring and combining the partial sums from each process. This is not a benchmark algorithm which flatters MPI, since it does not contain enough work to show speedup on most parallel machines; this is an algorithm which has been shown to be useful for comparing and contrasting programming environments.

For the collective communication performance tests, the implementations will use a reduction mechanism to have all processes reduce the partial sums in their thread block locally, and then have the processes pass their reduced partial sums to process 0. Process 0 will then complete the summation and return the result. Figure 5.17 shows the kernel for the algorithm implemented in CUDA with MPI, while Figure 5.18 and Figure 5.19 show the kernels for the algorithm implemented in CUDA without MPI.

```
/* MPI Implementation using collective communication */
__global__ void
PI__collective(register volatile int *interval_p,
                register volatile float *sum_p,
                register volatile int *error_p)
{
      register float width;
      __shared__ float sum;
      __shared__ float lsum[NUM_THREADS_PER_BLOCK];
      register int intervals, i;
      int nproc, iproc, ib;
      register int root = 0;

      /* Initialize MPI */
      if (MPI_Init((void *) 0) != MPI_SUCCESS) {
            error_p[0] = 1; return;
      }

      MPI_Comm_size(MPI_COMM_WORLD, &nproc);
      MPI_Comm_rank(MPI_COMM_WORLD, &iproc);
      if(iproc == 0) { root = MPI_ROOT; }

      intervals = *interval_p; width = 1.0 / intervals;
      ib = iproc % NUM_THREADS_PER_BLOCK; lsum[ib] = 0;
      for (i=iproc; i<intervals; i+=nproc) {
            register float x = (i + 0.5) * width;
            lsum[ib] += 4.0 / (1.0 + x * x);
      }
      lsum[ib] *= width;

      MPI_Reduce((void *) &(lsum[ib]), (void *) &(sum), 1,
                  MPI_FLOAT, MPI_SUM, root, MPI_COMM_WORLD);

      if(iproc == 0) { sum_p[0] = sum; }

      MPI_Finalize(); error_p[iproc] = 0;
      return;
}
```

Figure 5.17: Collective MPI implementation of the test algorithm

```
/* Native CUDA using collective communication */
/* Kernel 1 of 2 */
__global__ void
PI__collective_1(register volatile int *interval_p,
                 register volatile float *sum_p)
{
      register float width;
      __shared__ float lsum[NUM_THREADS_PER_BLOCK];
      register int intervals, i;
      int nproc, iproc, ib;
      __shared__ float sdata[NUM_THREADS_PER_BLOCK];

      nproc = blockDim.x * gridDim.x;
      iproc = (blockIdx.x * blockDim.x) + threadIdx.x;

      intervals = *interval_p; width = 1.0 / intervals;
      ib = iproc % NUM_THREADS_PER_BLOCK; lsum[ib] = 0;
      for (i=iproc; i<intervals; i+=nproc) {
            register float x = (i + 0.5) * width;
            lsum[ib] += 4.0 / (1.0 + x * x);
      }
      lsum[ib] *= width; __syncthreads();

      sdata[ib] = lsum[ib]; __syncthreads();

      if (ib <  64) { sdata[ib] += sdata[ib +  64]; }
      __syncthreads();
      if (ib < 32) { sdata[ib] += sdata[ib + 32]; }
      if (ib < 16) { sdata[ib] += sdata[ib + 16]; }
      if (ib < 8) { sdata[ib] += sdata[ib +  8]; }
      if (ib < 4) { sdata[ib] += sdata[ib +  4]; }
      if (ib < 2) { sdata[ib] += sdata[ib +  2]; }
      if (ib == 0) { sdata[ib] += sdata[ib +  1]; }

      if (ib == 0) { sum_p[blockIdx.x] = sdata[0]; }
      return;
}
```

Figure 5.18: Collective CUDA-only implementation of the test algorithm, part 1

```
/* Native CUDA using point-to-point communication */
/* Kernel 2 of 2 */
PI__collective_2(register volatile float *sum_p,
                 register volatile int *error_p)
{
      int iproc, ib;
      __shared__ float sdata[NUM_THREADS_PER_BLOCK];

      iproc = (blockIdx.x * blockDim.x) + threadIdx.x;

      ib = iproc % NUM_THREADS_PER_BLOCK;

      if (ib < 32) { sdata[ib] = sum_p[ib]; }

      if (ib < 16) { sdata[ib] += sdata[ib + 16]; }
      if (ib < 8) { sdata[ib] += sdata[ib +  8]; }
      if (ib < 4) { sdata[ib] += sdata[ib +  4]; }
      if (ib < 2) { sdata[ib] += sdata[ib +  2]; }
      if (ib == 0) { sdata[ib] += sdata[ib +  1]; }

      if (iproc == 0) { sum_p[0] = sdata[0]; }

      error_p[iproc] = 0;
      return;
}
```

Figure 5.19: Collective CUDA-only implementation of the test algorithm, part 2

The above codes require a little more explanation. The MPI implementation uses standard MPI functions to initialize, get the process rank and number of processes, and have every process perform a reduction operation, specifying process 0 as the root. Process 0 returns the final estimate to the host, and the computations are again nearly identical to those in the original algorithm, with the exception that they are performed in a shared-memory block. The native CUDA code uses CUDA built-ins to identify the process and number of processes, and performs the same computation as the MPI implementation. It then allocates a shared-memory block and performs an unrolled reduction operation in the shared-memory block, synchronizing only when necessary. Once the partial reduction is complete, the code uses a global memory location to pass out the partial sums for the next kernel invocation. The second CUDA kernel recalculates the process parameters and then has the low 32 processes in each block read the global memory into shared memory. Once read, the processes perform a tree reduction, just like

the first CUDA code, until process 0 writes the final solution out to global memory and all processes return.

To test the execution-time performance of these kernels, each was executed three times with the number of intervals per process varied from 2 to 128 in steps of 2. The resulting average execution times are shown in Figure 5.20.

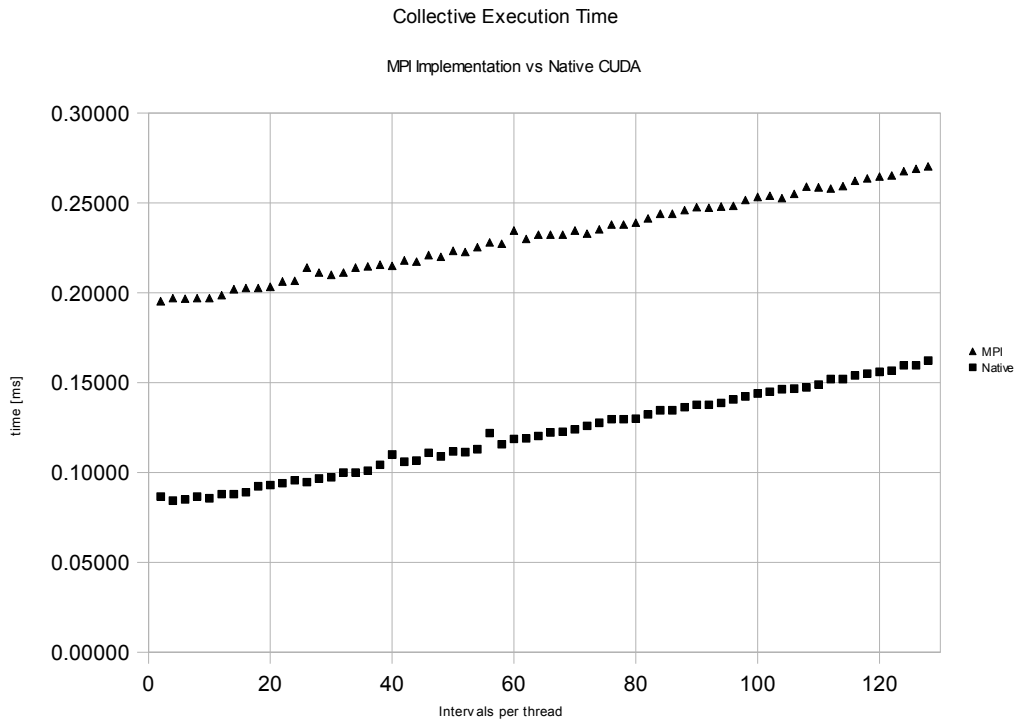

Figure 5.20: Execution time of both implementations of the test algorithm

The above chart shows the execution time for the MPI implementation and the native CUDA implementation. The average execution time of the MPI kernel is 0.231ms, while the average execution time of the native CUDA kernel is 0.121ms. The difference in execution time between 128 intervals per process and 2 intervals per process is 0.075ms for the MPI implementation and 0.076ms for the native CUDA implementation.

The performance data above indicates that there is roughly a 1.9x execution time performance penalty for using the MPI kernel over the native kernels with a collective communication implementation of the PI-estimation algorithm. It should first be noted that the increase in execution time from 2 intervals to 8 intervals per process, for both the MPI implementation and the native CUDA implementation, is nearly identical to the

increase in execution time of the native CUDA implementation performing point-to-point communication (0.079ms). This indicates that this is roughly the time required for the computation (the MPI point-to-point implementation hid this execution time behind global memory latency). This means that, for the native CUDA implementation, about 0.087ms is spent in communication. For the MPI implementation, about 0.19ms is spent in communication.

There are two reasons for the difference. The first, and most obvious reason is that the MPI implementation is not as efficient as the native CUDA implementation. On the second pass, for instance, the MPI implementation performs a linear reduction compared to the CUDA implementation's tree reduction. The MPI implementation also uses an aligned-to-misaligned copy routine to write its final value to the receive buffer in the root process. The second difference, which is not obvious at all, is the overhead in calling `MPI_Init`. The initialization routine must prepare the MPI system in its entirety, and this includes initializing all system data structures, whether used in the program or not. In particular, the initialization routine in the algorithm above initialized the RMA structures for one-sided communication and the message structures for point-to-point communication. The execution time performance gain observed by disabling these initializations is approximately 0.07ms. If this overhead could be removed by having the host initialize the structures, then the average time of the MPI implementation would be approximately 0.16ms.

In this case, the message-passing implementation is only slightly slower than the native CUDA implementation, but again provides a well-established and accepted interface. There is also a large performance benefit in terms of simpler, more-portable code, and the shared-memory issues are once again hidden nicely behind the MPI calls.

5.7: Other Implemented Interfaces

There are a handful of other interfaces which are implemented. These are either utility functions or other functions which can provide improved performance, and both these categories are described below.

The utility functions include the communication context management functions and the environmental management and inquiry functions. The 2 communication context

management interfaces implemented are `MPI_Comm_size` and `MPI_Comm_rank`, and the 3 environmental management and inquiry interfaces implemented are `MPI_Get_version`, `MPI_Init`, and `MPI_Finalize`.

`MPI_Comm_size` and `MPI_Comm_rank` are the process identification functions, and work on built-in CUDA variables. They have little overhead. `MPI_Get_version` also works on simple built-ins and has little overhead. `MPI_Finalize` is normally expensive because it implies a barrier synchronization, but it relies on the kernel stop as a synchronization mechanism in the implementation. `MPI_Init`, as discussed above, is not low-overhead due to its initialization of global data structures and requirement of a barrier synchronization. This overhead may be removable with more research.

There are 8 other functions are implemented because they had the potential to improve performance. These include 5 one-sided communication interfaces (`MPI_Win_create`, `MPI_Win_free`, `MPI_Put`, `MPI_Get`, `MPI_Win_fence`), and 3 data-type management interfaces (`MPI_Pack`, `MPI_Unpack`, `MPI_Pack_size`). The `MPI_Pack` and `MPI_Unpack` data-type management functions focus on packing data elements into buffers, and essentially perform misaligned-to-misaligned copy routines. `MPI_Pack_size` quickly calculates how much space a given call to `MPI_Pack` would require. The one-sided communication interfaces were originally interesting because it seemed that they could possibly help in hiding the strange semantics of the memory systems in the GPU. `MPI_Put` and `MPI_Get`, for example, allow a process to write to another processes memory space, and this could be very useful. The issue, though, is the shared-memory restriction. One-sided communication calls are only useful for processes in the same thread block with access to the same local shared memory, and this ruins most of the potential utility of these functions. They can, however, still be used to hide global-memory writes if needed, and once a window is created using `MPI_Win_create`, the accesses to the window are less-expensive than a corresponding send and receive simply because no handshaking or data-buffering is required in global memory.

5.8: Costs and Benefits of the Message-Passing Model

As clearly illustrated in the previous sections, the message-passing implementation has both costs and benefits. This section is not concerned with recapping those benefits, but rather intends to clarify and expand the costs and benefits beyond collective communications or point-to-point communications only.

The primary claim of this thesis is that the message-passing model can be conceptually cleaner by hiding, rather than dealing with, the quirks in underlying shared-memory models. The message-passing environment provides shorter code by hiding the piles of code needed to accomplish most tasks behind a clean, well-defined, and well-specified interface. The environment also results in fewer user-written operations, since many operations are provided by the MPI specification. The message-passing environment hides many GPU-dependent features such as owner-writes rules behind its standard interface, and also hides the global-memory semantics issues mentioned earlier. Finally, the message-passing implementation can result in better performance transparency, since the interfaces specified by MPI can be profiled cleanly and separately on any given GPU. Given the above factors, we suggest that the message-passing model is a significantly cleaner interface.

The message-passing model also has some disadvantages on the GPU. Virtualization is constrained by the MPI implementation because the underlying environment can hang (the lack of coherence issue again) if it is not. The MPI implementation also results in a larger code size, and current GPUs have a fixed limit for the number of instructions in a kernel. This is especially complicated by the fact that many current GPUs in-line functions, rather than calling them, which results in duplicated code if the same function is invoked multiple times. Finally, implementing a message-passing model on modern GPU hardware is not trivial, but performance suffers only slightly with careful choice of data structures and algorithms.

Given the trade-offs involved, we suggest that the prototype implementation of MPI discussed in this thesis can and does provide a conceptually cleaner interface for high-performance computing within a GPU.

Chapter 6: Conclusions and Future Work

In conclusion, this thesis has demonstrated that the message-passing programming model can be conceptually cleaner than the data-parallel model for programming GPUs. This provides a performance benefit by hiding any oddities with current shared-memory environments and also abstracting away GPU-specific features, while providing an interface which is well-established and well-understood. This thesis has also demonstrated that the virtualization constraint required by MPI within the GPU is harmless and compatible with any virtualization which is already optimal in terms of a strong interaction model and nearly-optimal per-block execution time.

The future work for this research will focus on first optimizing and thoroughly testing the existing implementation. This will involve profiling the individual parts carefully to determine which functions should be revisited. Also, many of the optimizations mentioned in the MPI implementation can clearly be applied and tested to improve the implementation. Finally, this work must be expanded onto other GPUs to determine its broader applicability, and also to determine what virtualization points will yield optimal behavior on different GPU systems.

Appendix A: CUDA Code for the Functions

```
/*************** MPI_Send ********************/

/* Send, standard-mode */
__device__ int MPI_Send (
      void* buf,              /* IN */
      int count,              /* IN */
      MPI_Datatype data-type, /* IN */
      int dest,               /* IN */
      int tag,                /* IN */
      MPI_Comm comm           /* IN */
)
{
      /* Declare any necessary variables */
      register int msg_slot = 0;
      register int msg_count = 0;
      register int i = 0;
      register int temp_a, temp_b;
      register int *buf_aligned;
      register char *temp_buf;

      /* Check arguments for errors */
      /* 1) Invalid dest -> Send to any
         2) Invalid dest -> Send to invalid processor number and
            not MPI_PROC_NULL
         3) Invalid tag -> >Upper Bound or <Lower Bound
         4) Invalid comm -> Not a supported communicator
         5) Invalid data-type -> not a supported type
         6) Invalid count -> negative or otherwise invalid
       */

      #if ENABLE_ERROR_CHECKING

      /* Check for error conditions */
      if( ((dest < 0) | (dest > (NPROC - 1))) &
            (dest != MPI_PROC_NULL)
      ) return (MPI_ERR_RANK);
      if( (tag < 0) | (tag > MPI_TAG_UB) | (tag == MPI_ANY_TAG)
      ) return (MPI_ERR_TAG);
      if( (comm != MPI_COMM_WORLD) ) return (MPI_ERR_COMM);
      if( (((data-type & MPI_UNSIGNED_TYPE)==1) &
            ((data-type & MPI_FLOAT_TYPE)== 1)) |
            (TYPE_SIZE(data-type) > MPI_LARGEST_TYPE) |
            (((TYPE_SIZE(data-type))&((TYPE_SIZE(data-type))-1))!=0)
      ) return(MPI_ERR_TYPE);
      if( (count <= 0)|
            ((count*TYPE_SIZE(data-type))>(MAX_DATA_PER_MESSAGE<<2))
      ) return (MPI_ERR_COUNT);

      #endif

      /* Special case for send to MPI_PROC_NULL */
      if(dest == MPI_PROC_NULL) return(MPI_SUCCESS);

      /* Reset received buffers as available and adjust the total
```

```
 *    buffered message count accordingly */
msg_count = PE_MSG_COUNT(IPROC);
i=0;
while ( msg_count > 0 )
{
      switch( PE_MSG_SERIALS(IPROC,i) )
      {
            case SN_AVAILABLE:
                  break;
            case SN_RECEIVED:
                  PE_MSG_SERIALS(IPROC,i) = SN_AVAILABLE;
                  --PE_MSG_COUNT(IPROC);
            default:
                  --msg_count;
      }
      ++i;
}

/* Ensure that buffer space is still available */
if (PE_MSG_COUNT(IPROC) == MAX_BUFFERED_MESSAGES)
      return (MPI_ERR_NO_SPACE);

/* Now find the first free buffer (one marked as available) */
msg_slot=-1;
do{   i = PE_MSG_SERIALS(IPROC,++msg_slot);
} while(i != SN_AVAILABLE);

/* Fill the buffer with the message header */
MSG_DST(IPROC, msg_slot) = dest;
MSG_DTYPE(IPROC, msg_slot) = data-type;
MSG_COUNT(IPROC, msg_slot) = count;
MSG_TAG(IPROC, msg_slot) = tag;
MSG_COMM(IPROC, msg_slot) = comm;

/* Copy the data from the send buffer to the the system buffer */
/* (This copy actually copies more data than necessary, but
 * reading extra bytes is not problematic since the GPU
 * allocates memory on word-boundaries anyway) */

/* Get the aligned source address */
/*    This is actually just:
 *    buf_aligned = ((int *) (((long) buf) & (~3)));
 *    but nvcc doesn't realize that this won't change a
 *    pointer's domain (global or shared mem), and so it forces
 *    the pointer to global mem only.  The following code is the
 *    same pointer math modified for the nvcc compiler. */
switch(((long) buf) & 3)
{

case 3:     temp_buf =  ((char *) buf) - 3;
            break;

case 2:     temp_buf = ((char *) buf) - 2;
            break;

case 1:     temp_buf = ((char *) buf) - 1;
            break;
```

```
case 0:     temp_buf = (char *) buf;
            break;
}
buf_aligned = (int *) temp_buf;

/* Convert the count to bytes */
count = count * TYPE_SIZE(data-type);

/* The read is based on the misalignment of the source */
i=0;
switch(((long) buf) & 3) {

case 0:     /* The source is aligned on a word boundary */
      while(count > 0)
      {
            MSG_DATA(IPROC,msg_slot,i++) = *(buf_aligned++);
            count -= 4;
      }
      break;

case 1:     /* The source is misaligned by one byte */
      temp_a = *(buf_aligned++);
      while (count > 0)
      {
            temp_a >>= 8;
            temp_b = *(buf_aligned++);
            temp_a = ((temp_a & 0x00ffffff) | (temp_b << 24));
            MSG_DATA(IPROC,msg_slot,i++) = temp_a;
            temp_a = temp_b;
            count -= 4;
      }
      break;

case 2:     /* The source is misaligned by two bytes */
      temp_a = *(buf_aligned++);
      while (count > 0)
      {
            temp_a >>= 16;
            temp_b = *(buf_aligned++);
            temp_a = ((temp_a & 0x0000ffff) | (temp_b << 16));
            MSG_DATA(IPROC,msg_slot,i++) = temp_a;
            temp_a = temp_b;
            count -= 4;
      }
      break;

case 3:     /* The source is misaligned by three bytes */
      temp_a = *(buf_aligned++);
      while (count > 0)
      {
            temp_a >>= 24;
            temp_b = *(buf_aligned++);
            temp_a = ((temp_a & 0x000000ff) | (temp_b << 8));
            MSG_DATA(IPROC,msg_slot,i++) = temp_a;
            temp_a = temp_b;
            count -= 4;
```

```
            }
            break;
      }

      /* Finally, increment the current message count */
      ++PE_MSG_COUNT(IPROC);

      /* Serialize the message, which marks it as valid. */
      PE_MSG_SERIALS(IPROC, msg_slot) = PE_SERIAL(IPROC);
      ++PE_SERIAL(IPROC);

      /* Return */
      return (MPI_SUCCESS);
}


/*************** MPI_Recv ********************/

/* Receive */
__device__ int
MPI_Recv
(
      void* buf,              /* OUT */
      int count,              /* IN */
      MPI_Datatype data-type, /* IN */
      int source,             /* IN */
      int tag,                /* IN */
      MPI_Comm comm,          /* IN */
      MPI_Status *status      /* OUT */
)
{

      /* Declare any necessary variables */
      register int msg_slot = -1;
      register int msg_count = 0;
      register int i = 0;
      register int serial = (1 << 30);
      register int temp_a, temp_b;
      register int *buf_aligned;
      register char *temp_buf;

      /* Check for error conditions */
      /* 1) Invalid source -> Recv from invalid processor number and
            not MPI_PROC_NULL.
         3) Invalid tag -> >UB or <LB and not MPI_ANY_TAG
         4) Invalid comm -> Not a supported communicator
         6) Invalid data-type -> not a supported type
         5) Invalid count -> negative or otherwise invalid
       */

      #if ENABLE_ERROR_CHECKING

      /* Check arguments for errors */
      if( ((source < 0) | (source > (NPROC - 1))) &
            (source != MPI_PROC_NULL)
      ) return (MPI_ERR_RANK);
      if( ((tag < 0) | (tag > MPI_TAG_UB)) & (tag != MPI_ANY_TAG)
```

```
) return (MPI_ERR_TAG);
if( (count <= 0) |
      ((count*TYPE_SIZE(data-type))>(MAX_DATA_PER_MESSAGE << 2))
) return (MPI_ERR_COUNT);
if( (comm != MPI_COMM_WORLD) ) return (MPI_ERR_COMM);
if( (TYPE_SIZE(data-type) > MPI_LARGEST_TYPE) |
      (((TYPE_SIZE(data-type))&((TYPE_SIZE(data-type))-1))!=0) |
      (((data-type & MPI_UNSIGNED_TYPE) == 1) &
      ((data-type & MPI_FLOAT_TYPE) == 1))
) return(MPI_ERR_TYPE);

#endif

/* Special case for recv from MPI_PROC_NULL */
if(source == MPI_PROC_NULL)
{
      /* Set the status object and return. */
      (*status).MPI_SOURCE = MPI_PROC_NULL;
      (*status).MPI_TAG = MPI_ANY_TAG;
      (*status).recv_count = 0;
      return(MPI_SUCCESS);
}

/* This is a blocking receive operation, so
 *    loop until a matching send is found */
msg_slot = -1;
while( msg_slot == -1 )
{
      msg_count = PE_MSG_COUNT(source);
      i = 0;

      /* Envelope matching */
      while( ( msg_count > 0) & (i < MAX_BUFFERED_MESSAGES) )
      {
            if(PE_MSG_SERIALS(source, i) != SN_AVAILABLE)
            {
                  --msg_count;

                  /* If the serial number is valid, check the
                   * envelope of this message */
                  if((PE_MSG_SERIALS(source, i) != SN_RECEIVED) &
                        (PE_MSG_SERIALS(source, i) < serial) &
                        (MSG_DST(source, i) == IPROC) &
                        (MSG_TAG(source, i) == tag) &
                        (MSG_COMM(source, i) == comm)
                  ) {
                        msg_slot = i;
                        serial = PE_MSG_SERIALS(source, i);
                  }
            }

            ++i;
      }
}

/* We now have a matching message slot. */
```

```
/* The message will now be consumed even if the call generates an
 * error, so the status object is set. */
(*status).MPI_SOURCE = source;
(*status).MPI_TAG = tag;
(*status).recv_count = MSG_COUNT(source,msg_slot);

/* Check that the data-types match */
switch(data-type)
{
      case MPI_PACKED:
            break;

      default:
            if( data-type != MSG_DTYPE(source,msg_slot) )
            {
                  PE_MSG_SERIALS(source, msg_slot) = SN_RECEIVED;
                  return(MPI_ERR_TYPE);
            }
            break;
}

/* Ensure that recv buffer can hold message */
if( count < MSG_COUNT(source, msg_slot) )
{
      PE_MSG_SERIALS(source, msg_slot) = SN_RECEIVED;
      return(MPI_ERR_TRUNCATE);
}

/* Copy the data from the system buffer to the the recv buffer */

/* Get the aligned source address */
/*    This is actually just:
 *    buf_aligned = ((int *) (((long) buf) & (~3)));
 *    but nvcc doesn't realize that this won't change a
 *    pointer's domain (global or shared mem), and so it forces
 *    the pointer to global mem only.  The following code is the
 *    same pointer math modified for the nvcc compiler. */
switch(((long) buf) & 3)
{

case 3: temp_buf =  ((char *) buf) - 3;
      break;

case 2: temp_buf = ((char *) buf) - 2;
      break;

case 1: temp_buf = ((char *) buf) - 1;
      break;

case 0: temp_buf = (char *) buf;
      break;

}
buf_aligned = (int *) temp_buf;

/* Convert the count to bytes */
count = MSG_COUNT(source, msg_slot) * TYPE_SIZE(data-type);
```

```
/* The read is based on the misalignment of the destination */
i=0;
switch(((long) buf) & 3) {

case 0:     /* The destination buffer is aligned */
      while (count > 3)
      {
            *(buf_aligned++) = MSG_DATA(source,msg_slot,i++);
            count -= 4;
      }
      temp_a = MSG_DATA(source,msg_slot,i);
      switch(count)
      {
            case 3: *(((char *) buf_aligned) + 2)=(temp_a >> 16);
            case 2: *(((char *) buf_aligned) + 1)=(temp_a >> 8);
            case 1: *((char *) buf_aligned)=temp_a;
            case 0: ;
      }
      break;

case 1:     /* The destination buffer is misaligned by one byte */
      temp_b = MSG_DATA(source,msg_slot,i++);
      switch(count)
      {
            default:
            case 3: *(((char *) buf_aligned) + 3)=(temp_b >> 16);
                  --count;
            case 2: *(((char *) buf_aligned) + 2)=(temp_b >> 8);
                  --count;
            case 1:     *(((char *) buf_aligned) + 1)=temp_b;
                  --count;
      }

      ++buf_aligned;

      /* Now that the destination is aligned, assemble messages */
      while(count > 3)
      {
            temp_b >>= 24;
            temp_a = MSG_DATA(source,msg_slot,i++);
            temp_b = ((temp_a << 8) | (temp_b & 0x000000ff));
            *(buf_aligned++) = temp_b;
            temp_b = temp_a;
            count -= 4;
      }

      temp_a = MSG_DATA(source,msg_slot,i);
      switch(count)
      {
            case 3: *(((char *) buf_aligned) + 2)=(temp_a >> 8);
            case 2: *(((char *) buf_aligned) + 1)=(temp_a);
            case 1: *((char *) buf_aligned)=(temp_b >> 24);
            case 0: ;
      }
      break;
```

```
case 2:     /* The destination buffer is misaligned by 2 bytes */
      temp_b = MSG_DATA(source,msg_slot,i++);
      switch(count)
      {
            default:
            case 2: *(((char *) buf_aligned) + 3)=(temp_b >> 8);
                  --count;
            case 1:     *(((char *) buf_aligned) + 2)=temp_b;
                  --count;
      }

      ++buf_aligned;

      /* Now that the destination is aligned, assemble messages */
      while(count > 3)
      {
            temp_b >>= 16;
            temp_a = MSG_DATA(source,msg_slot,i++);
            temp_b = ((temp_a << 16) | (temp_b & 0x0000ffff));
            *(buf_aligned++) = temp_b;
            temp_b = temp_a;
            count -= 4;
      }

      temp_a = MSG_DATA(source,msg_slot,i);
      switch(count)
      {
            case 3: *(((char *) buf_aligned) + 2)=(temp_a);
            case 2: *(((char *) buf_aligned) + 1)=(temp_b >> 24);
            case 1: *((char *) buf_aligned)=(temp_b >> 16);
            case 0: ;
      }
      break;

case 3:     /* The destination buffer is misaligned by 3 bytes */
      temp_b = MSG_DATA(source,msg_slot,i++);

      /* The count is always at least one */
      *(((char *) buf_aligned) + 3) = temp_b;
      --count;

      ++buf_aligned;

      /* Now that the destination is aligned, assemble messages */
      while(count > 3)
      {
            temp_b >>= 8;
            temp_a = MSG_DATA(source,msg_slot,i++);
            temp_b = ((temp_a << 24) | (temp_b & 0x00ffffff));
            *(buf_aligned++) = temp_b;
            temp_b = temp_a;
            count -= 4;
      }

      switch(count)
      {
            case 3: *(((char *) buf_aligned) + 2)=(temp_b >> 24);
```

```
                case 2: *(((char *) buf_aligned) + 1)=(temp_b >> 16);
                case 1: *((char *) buf_aligned)=(temp_b >> 8);
                case 0: ;
            }
            break;
      }

      /* Mark the message as received */
      PE_MSG_SERIALS(source, msg_slot) = SN_RECEIVED;

      /* Return */
      return(MPI_SUCCESS);
}


/*************** MPI_Barrier ********************/

/* Barrier synchronization */
/* (Uses a single-writer, multiple-reader mechanism in global memory)*/
__device__ int
MPI_Barrier
(
      MPI_Comm comm           /* IN */
)
{
      /* This is a barrier synchronization using a single-writer,
       * multiple-reader mechanism in global memory. */

      /* Declare any necessary variables */
      register int err_code = MPI_SUCCESS;

      /* Check for error conditions */
      /* 1) Invalid comm -> Not a supported communicator
       */

      #if ENABLE_ERROR_CHECKING

      /* Check arguments for errors */
      if( (comm != MPI_COMM_WORLD) ) err_code = (MPI_ERR_COMM);

      #endif

      /* Get a pointer (in a register) to the volatile global memory
       * segment where the barrier numbers reside. */
      register volatile int *bar_nums = &(system_barrier_buffer[0]);

      /* Sync here to be sure that everybody in this block is at the
       * barrier */
      __syncthreads();

      /* The first thread in the block represents the whole block in
       * the synchronization */
      if (IPROC_IN_BLOCK == 0)
      {
            /* We need the next storage for the next block's
             * barrier number */
            register int his_bar_num;
```

```
            /* Get the current barrier number for this block */
            register int my_bar_num = bar_nums[BIPROC] + 1;

            /* This is the starting location for the scan */
            register int i = ((BIPROC + 1) % BNPROC);

            /* Increment this block's barrier number */
            bar_nums[BIPROC] = my_bar_num;

            /* Now, starting at the next block, scan the barrier
             * numbers until either all other blocks have arrived or
             * one block has passed. */
            do {

                  /* Wait for the block to arrive */
                  do
                  {
                        his_bar_num = bar_nums[i];
                  } while (his_bar_num < my_bar_num);

                  /* Get the next index, wrapping if needed */
                  if (++i >= BNPROC) i = 0;

            } while ((his_bar_num == my_bar_num) & (i != BIPROC));

            /* The barrier is completed. Increment again to inform
             * anybody still scanning */
            bar_nums[BIPROC] = my_bar_num + 1;

      }

      /* All threads wait here for the first thread */
      __syncthreads();

      /* Return */
      return(err_code);
}


/*************** MPI_Reduce ******************/

/* Reduction */
/* This macro will create a reduction function working on 4-byte
 * objects.  The count modifier count_mod could be used when working
 * with short or char types SWAR-style */
#define CREATE_FUNCTION_MPI_REDUCE_4_BYTE_OPS(op,
      op,op_symbol,dtype,dtype_symbol,ident_val) \
__device__ int \
MPI_Reduce_##dtype##_##op \
( \
      void* sendbuf,    /* IN */ \
      void* recvbuf,    /* OUT */ \
      int count,        /* IN */ \
      int root,         /* IN */ \
      MPI_Comm comm     /* IN */ \
) \
```

```
{ \
\
      /* Declare any necessary variables */ \
      register int temp_a, temp_b, temp_count, i; \
      register int err_code = MPI_SUCCESS; \
      register int *recvbuf_aligned; \
      register char *temp_buf; \
      __shared__ dtype_symbol shared_data[NPROC_IN_BLOCK]; \
\
\
      /* Check arguments for errors */ \
      /* 1) Invalid sendbuf -> null \
         2) Invalid recvbuf -> null \
         3) Invalid count -> negative, zero, or greater than the max \
         4) Invalid comm -> Not a supported communicator \
       */ \
\
      if(ENABLE_ERROR_CHECKING) { \
\
      /* Check for error conditions */ \
      if( ((root < 0) | (root > (NPROC - 1))) & (root != MPI_ROOT) & \
            (root != MPI_PROC_NULL) \
      ) err_code = (MPI_ERR_ROOT); \
      if( ((((long) sendbuf) & 3) != 0) ) err_code = (MPI_ERR_BUFFER); \
      if( (count <= 0) | \
            ((count*TYPE_SIZE(dtype)) > \
            ((MAX_DATA_PER_MESSAGE *MAX_BUFFERED_MESSAGES) << 2)) \
      ) err_code = (MPI_ERR_COUNT); \
      if( (((count) & (count - 1)) != 0) ) err_code = (MPI_ERR_COUNT); \
      if( (comm != MPI_COMM_WORLD) ) err_code =  (MPI_ERR_COMM); \
\
      } \
\
      /* For each element, have all processes read that element into \
       *    shared mem from their sendbuf and reduce the values in \
       *    shared mem, then have the first process in each block \
       *    write the value out into global memory. */ \
\
      for(temp_count = 0 ; temp_count < count ; ++temp_count) \
      { \
            /* First, copy the data to the the shared memory block */ \
            shared_data[IPROC_IN_BLOCK] = ((err_code == MPI_SUCCESS) ? \
            ((dtype_symbol *) sendbuf)[temp_count] : (ident_val) ); \
            __syncthreads(); \
\
            /* Now, perform the tree reduction in each block */ \
            for(i = (NPROC_IN_BLOCK >> 1) ; i > 0 ; i >>= 1) \
            { \
                  if(IPROC_IN_BLOCK < i) \
                  { \
                        if((op != MPI_MIN) & (op != MPI_MAX)) \
                        { \
                              shared_data[IPROC_IN_BLOCK] = \
                              shared_data[IPROC_IN_BLOCK] op_symbol \
                              shared_data[IPROC_IN_BLOCK + i]; \
                        } \
                        if((op == MPI_MIN) | (op == MPI_MAX)) \
```

```
                        { \
                              shared_data[IPROC_IN_BLOCK] = \
                              ( (shared_data[IPROC_IN_BLOCK] op_symbol \
                              shared_data[IPROC_IN_BLOCK + i]) ? \
                              shared_data[IPROC_IN_BLOCK] : \
                              shared_data[IPROC_IN_BLOCK + i] ); \
                        } \
                  } \
                  __syncthreads(); \
            } \
\
            /* Now write the data into the sys buffer for this block*/ \
            if(IPROC_IN_BLOCK == 0) \
            { \
                  MSG_DATA(IPROC, 0, temp_count) = \
                  *((datum_t *) &(shared_data[0])); \
            } \
      } \
\
      /* Use a barrier synchronization to ensure that all values are \
       * written in global memory */ \
      MPI_Barrier(comm); \
\
      /* Do a linear reduction using one process per element, since \
       * there are not many blocks. */ \
\
      /* Let all the blocks reduce to a final answer. */ \
      if(IPROC_IN_BLOCK < BNPROC) \
      { \
            shared_data[IPROC_IN_BLOCK] = \
            *((dtype_symbol *) &MSG_DATA(0, 0, IPROC_IN_BLOCK)); \
\
            for(temp_count = NPROC_IN_BLOCK ; \
                  temp_count < NPROC ; \
                  temp_count += NPROC_IN_BLOCK) \
            { \
                  if((op != MPI_MIN) & (op != MPI_MAX)) \
                  { \
                        shared_data[IPROC_IN_BLOCK] = \
                        shared_data[IPROC_IN_BLOCK] op_symbol \
                        *((dtype_symbol *) \
                        &MSG_DATA(temp_count, 0, IPROC_IN_BLOCK)); \
                  } \
                  if((op == MPI_MIN) | (op == MPI_MAX)) \
                  { \
                        shared_data[IPROC_IN_BLOCK] = \
                        ( (shared_data[IPROC_IN_BLOCK] op_symbol \
                        *((dtype_symbol *) \
                        &MSG_DATA(temp_count, 0, IPROC_IN_BLOCK))) ? \
                        shared_data[IPROC_IN_BLOCK] : \
                        *((dtype_symbol *) \
                        &MSG_DATA(temp_count, 0, IPROC_IN_BLOCK)) ); \
                  } \
            } \
      } \
\
      /* Synchronize root and read the data */ \
```

```
        __syncthreads(); \
        if(root == MPI_ROOT) \
        { \
              /* Copy the data from shared memory to the the buffer */ \
\
              /* Get the aligned recv buffer address */ \
              /* (The long version of recvbuf_aligned = \
               * ((int *) (((long) recvbuf) & (~3)));) */ \
              switch(((long) recvbuf) & 3) \
              { \
\
              case 3: temp_buf =  ((char *) recvbuf) - 3; \
                    break; \
\
              case 2: temp_buf = ((char *) recvbuf) - 2; \
                    break; \
\
              case 1: temp_buf = ((char *) recvbuf) - 1; \
                    break; \
\
              case 0: temp_buf = (char *) recvbuf; \
                    break; \
\
              } \
              recvbuf_aligned = (int *) temp_buf; \
\
              /* Convert the count to bytes */ \
              count = count * TYPE_SIZE(dtype); \
\
              /* The read is based on the misalignment of the dest */ \
              i=0; \
              switch(((long) recvbuf) & 3) { \
\
              case 0:     /* The destination buffer is aligned */ \
                    while (count > 3) \
                    { \
                          *(recvbuf_aligned++) = \
                                *((int *) &(shared_data[i++])); \
                          count -= 4; \
                    } \
                    temp_a = *((int *) &(shared_data[i])); \
                    switch(count) \
                    { \
                          case 3: *(((char *) recvbuf_aligned) + 2) = \
                                      (temp_a >> 16); \
                          case 2: *(((char *) recvbuf_aligned) + 1) = \
                                      (temp_a >> 8); \
                          case 1: *((char *) recvbuf_aligned) = temp_a; \
                          case 0: ; \
                    } \
                    break; \
\
              case 1:     /* The dest buffer is misaligned by 1 byte */ \
                    temp_b = *((int *) &(shared_data[i++])); \
                    switch(count) \
                    { \
                          default: \
```

```
                        case 3: *(((char *) recvbuf_aligned) + 3) = \
                              (temp_b >> 16); \
                              --count; \
                        case 2: *(((char *) recvbuf_aligned) + 2) = \
                              (temp_b >> 8); \
                              --count; \
                        case 1: *(((char *) recvbuf_aligned) + 1) = \
                              temp_b; \
                              --count; \
                  } \
\
                  ++recvbuf_aligned; \
\
                  /* Now that the dest is aligned, assemble messages*/ \
                  while(count > 3) \
                  { \
                        temp_b >>= 24; \
                        temp_a = *((int *) &(shared_data[i++])); \
                        temp_b = ((temp_a << 8)|(temp_b&0x000000ff)); \
                        *(recvbuf_aligned++) = temp_b; \
                        temp_b = temp_a; \
                        count -= 4; \
                  } \
\
                  temp_a = *((int *) &(shared_data[i])); \
                  switch(count) \
                  { \
                        case 3: *(((char *) recvbuf_aligned) + 2) = \
                              (temp_a >> 8); \
                        case 2: *(((char *) recvbuf_aligned) + 1) = \
                              (temp_a); \
                        case 1: *((char *) recvbuf_aligned) = \
                              (temp_b >> 24); \
                        case 0: ; \
                  } \
                  break; \
\
            case 2:     /* The dest buffer is misaligned by 2 bytes */ \
                  temp_b = *((int *) &(shared_data[i++])); \
                  switch(count) \
                  { \
                        default: \
                        case 2: *(((char *) recvbuf_aligned) + 3) = \
                              (temp_b >> 8); \
                              --count; \
                        case 1: *(((char *) recvbuf_aligned) + 2) = \
                              temp_b; \
                              --count; \
                  } \
\
                  ++recvbuf_aligned; \
\
                  /* Now that the dest is aligned, assemble messages*/ \
                  while(count > 3) \
                  { \
                        temp_b >>= 16; \
                        temp_a = *((int *) &(shared_data[i++])); \
```

```
                        temp_b = ((temp_a<<16)|(temp_b&0x0000ffff)); \
                        *(recvbuf_aligned++) = temp_b; \
                        temp_b = temp_a; \
                        count -= 4; \
                  } \
\
                  temp_a = *((int *) &(shared_data[i])); \
                  switch(count) \
                  { \
                        case 3: *(((char *) recvbuf_aligned) + 2) = \
                              (temp_a); \
                        case 2: *(((char *) recvbuf_aligned) + 1) = \
                              (temp_b >> 24); \
                        case 1: *((char *) recvbuf_aligned) = \
                              (temp_b >> 16); \
                        case 0: ; \
                  } \
                  break; \
\
            case 3:     /* The dest buffer is misaligned by 3 bytes */ \
                  temp_b = *((int *) &(shared_data[i++])); \
\
                  /* The count is always at least one */ \
                  *(((char *) recvbuf_aligned) + 3) = temp_b; \
                  --count; \
\
                  ++recvbuf_aligned; \
\
                  /* Now that the dest is aligned, assemble messages*/ \
                  while(count > 3) \
                  { \
                        temp_b >>= 8; \
                        temp_a = *((int *) &(shared_data[i++])); \
                        temp_b = ((temp_a<<24)|(temp_b&0x00ffffff)); \
                        *(recvbuf_aligned++) = temp_b; \
                        temp_b = temp_a; \
                        count -= 4; \
                  } \
\
                  switch(count) \
                  { \
                        case 3: *(((char *) recvbuf_aligned) + 2) = \
                              (temp_b >> 24); \
                        case 2: *(((char *) recvbuf_aligned) + 1) = \
                              (temp_b >> 16); \
                        case 1: *((char *) recvbuf_aligned) = \
                              (temp_b >> 8); \
                        case 0: ; \
                  } \
                  break; \
            } \
      } \
\
      /* Now, synchronize again to let root finish */ \
      MPI_Barrier(comm); \
\
      /* Return */ \
```

```
      return(err_code); \
} \
\

/* Declarations for the various MPI_Reduce_xxx_xxx() functions */

/* Floating point functions */
CREATE_FUNCTION_MPI_REDUCE_4_BYTE_OPS(MPI_SUM,+,MPI_FLOAT,float,0.0f)

/* Reduction */
/* This macro changes the MPI_Reduce() function call to a particular
 * (data-type,op) function variant at compile-time. */
#define MPI_Reduce(sendbuf,recvbuf,count,data-type,op,root,comm) \
      MPI_Reduce_##data-type##_##op(sendbuf,recvbuf,count,root,comm)

/* This macro creates a reduction function prototype. */
#define CREATE_PROTOTYPE_MPI_REDUCE_4_BYTE_OPS(op,dtype) \
__device__ int \
MPI_Reduce_##dtype##_##op \
( \
      void* sendbuf,    /* IN */ \
      void* recvbuf,    /* OUT */ \
      int count,        /* IN */ \
      int root,         /* IN */ \
      MPI_Comm comm     /* IN */ \
);

/* Declarations for the various MPI_Reduce_xxx_xxx() function
 * prototypes */

/* Floating point prototypes */
CREATE_PROTOTYPE_MPI_REDUCE_4_BYTE_OPS(MPI_SUM,MPI_FLOAT)
```

Bibliography

[1] Message Passing Interface Forum, “MPI: A Message-Passing Interface Standard,” Version 2.1, June 2008. [Online]. Available: http://www.mpi-forum.org/docs/mpi21-report.pdf. [Accessed: August 5, 2009].

[2] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," *ACM Queue*, vol. 6, no. 2, pp. 40-53, 2008.

[3] H. Richardson, “High Performance Fortran: history, overview, and current developments,” Version 1.4, Thinking Machines Corporation, Bedford, MA, USA, Tech. Rep. TMC-261, 1996.

[4] I. Buck, et al., “Brook for GPUs: Stream Computing on Graphics Hardware,” *ACM Transactions on Graphics*, vol. 23, pp. 777-786, 2004.

[5] ATI, “ATI Stream Computing – Technical Overview,” ATI Stream Developer Articles & Publications, March 2009. [Online]. Available: http://developer.amd.com/gpu_assets/Stream_Computing_Overview.pdf. [Accessed: August 5, 2009].

[6] E. Gabriel, et al., "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation,” in *Lecture Notes in Computer Science: Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Vol. 3241, D. Kranzlmuller, P. Kacsuk, J. Dongarra, Ed. Heidelberg, Germany: Springer Berlin, 2004, pp. 97-104.

[7] G. Burns, R. Daoud, and J. Vaigl, “LAM: An Open Cluster Environment for MPI,” in *Proceedings of Supercomputing Symposium*, 1994, pp. 379-386.

[8] M. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. 21, pp. 948-960, September 1972.

[9] F. Darema, “SPMD model: past, present and future,” in *Lecture Notes in Computer Science: Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Vol. 2131, Y. Cotronis, J. Dongarra, Ed. Heidelberg, Germany: Springer Berlin, 2001, pp. 1.

[10] L. Seiler, et al., "Larrabee: A many-core x86 architecture for visual computing," *IEEE Micro*, vol. 29, no. 1, pp. 10-21, 2009.

[11] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata, “Cell Broadband Engine Architecture and its First Implementation: A Performance View,” *IBM Journal of Research and Development*, vol. 51, no. 5, pp. 559-572, 2007.

[12] M. Rivas, “AMD Financial Analyst Day 2007 Presentation,” AMD, December 2007. [Online]. Available: http://download.amd.com/Corporate/MarioRivasDec2007AMDAnalystDay.pdf. [Accessed: August 5, 2009].

[13] R. Bergman, “AMD Financial Analyst Day 2008 Presentation,” AMD, November 2008. [Online]. Available: http://www.amd.com/us-en/assets/content_type/DownloadableAssets/RickBergmanAMD2008AnalystDay11-13-08.pdf. [Accessed: August 5, 2009].

[14] Microsoft Corporation, “DirectX Graphics,” Microsoft Developer Network, March 2009. [Online]. Available: http://msdn.microsoft.com/en-us/library/bb219740(VS.85).aspx. [Accessed: August 5, 2009].

[15] Khronos Group, “OpenGL 3.2 Core Profile Specification,” August 2009. [Online]. Available: http://www.opengl.org/registry/doc/glspec32.core.20090803.pdf. [Accessed: August 5, 2009].

[16] Microsoft Corporation, “HLSL,” Microsoft Developer Network, March 2009. [Online]. Available: http://msdn.microsoft.com/en-us/library/bb509561(VS.85).aspx. [Accessed: August 5, 2009].

[17] Khronos Group, “OpenGL Shading Language 1.50.09 Specification,” July 2009. [Online]. Available: http://www.opengl.org/registry/doc/GLSLangSpec.1.50.09.pdf. [Accessed: August 5, 2009].

[18] W. Mark, R. Glanville, K. Akeley, and M. Kilgard, “Cg: a System for Programming Graphics Hardware in a C-like Language,” *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 896-907, 2003.

[19] M. McCool, and S. Du Toit, *Metaprogramming GPUs with Sh*,” Wellesley, MA, USA: A K Peters, Ltd., 2004.

[20] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses," in *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 2006, pp. 325-335.

[21] M. Monteyne, “RapidMind Multi-Core Development Platform,” RapidMind Inc., Waterloo, Canada, February 2008. [Online]. Available: http://www.rapidmind.net/pdfs/WP_RapidMindPlatform.pdf. [Accessed: August 5, 2009].

[22] ATI, “ATI CTM Guide,” Version 1.01, November 2006. [Online]. Available: http://ati.amd.com/companyinfo/researcher/documents.html. [Accessed: August 5, 2009].

[23] NVIDIA, “NVIDIA CUDA Programming Guide,” Version 2.1, December 2008. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/2_1/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.1.pdf. [Accessed: August 5, 2009].

[24] M. Papakipos, “The PeakStream Platform: High-Productivity Software Development for Multi-Code Processors,” PeakStream Inc., Redwood City, CA, USA, April 2007. [Online]. Available: http://www.linuxclustersinstitute.org/conferences/archive/2007/PDF/papakipos_21367.pdf. [Accessed: August 5, 2009].

[25] ATI, “ATI Stream Computing User Guide,” Revision 1.4.0, March 2009. [Online]. Available: http://developer.amd.com/gpu_assets/Stream_Computing_User_Guide.pdf. [Accessed: August 5, 2009].

[26] Khronos OpenCL Working Group, “The OpenCL Specification,” Version 1.0, May 2009. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.0.43.pdf. [Accessed: August 5, 2009].

[27] M. Strengert, C. Miller, C. Dachsbacher, and T. Ertl, “CUDASA: Compute Unified Device and Systems Architecture,” in *Eurographics Symposium on Parallel Graphics and Visualization*, pp. 49-56, 2008.

[28] Z. Fan, F. Qiu, and A. Kaufman, “Zippy: A Framework for Computation and Visualization on a GPU Cluster,” *Computer Graphics Forum*, vol. 27, no. 2, pp. 341-350, April 2008.

[29] J. Stuart, and J. Owens, “Message Passing on Data-Parallel Architectures,” in *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, May 2009. [Online]. Available: http://www.idav.ucdavis.edu/func/return_pdf?pub_id=959. [Accessed: August 5, 2009].

[30] Q. Hou, K. Zhou, and B. Guo, “BSGP: Bulk-Synchronous GPU Programming,” *ACM Transactions on Graphics*, vol. 27, no. 3, August 2008.

[31] L. Tucker, G. Robertson, "Architecture and Applications of the Connection Machine,"*Computer*, vol. 21, no. 8, pp. 26-38, Aug. 1988.

[32] NVIDIA, “NVIDIA GeForce 8800 GT,” NVIDIA Products / Desktop / GeForce Family. [Online]. Available: http://www.nvidia.com/object/product_geforce_8800_gt_us.html. [Accessed: August 5, 2009].

[33] NVIDIA, “NVIDIA GeForce 9800 GT,” NVIDIA, Products / Desktop / GeForce Family / NVIDIA GeForce 9800 GT. [Online]. Available: http://www.nvidia.com/object/product_geforce_9800gt_us.html. [Accessed: August 5, 2009].

[34] NVIDIA, “GeForce GTX 280,” NVIDIA, Products / Desktop / GeForce Family. [Online]. Available: http://www.nvidia.com/object/product_geforce_gtx_280_us.html. [Accessed: August 5, 2009].

[35] J. Nickolls, “The Design of the MasPar MP-1: A Cost Effective Massively Parallel Computer,” *Thirty-Fifteh IEEE Computer Society International Conference: Intellectual Leverage*, pp. 25-28, February 1990.

[36] C. Whaley, A. Petitet, and J. Dongarra, “Automated Empirical Optimization of Software and the ATLAS Project,” *Parallel Computing*, vol. 27, 2001.

[37] H. Dietz, T. Mattox, and G. Krishnamurthy, “The Aggregate Function API: It's not just for PAPERS anymore,” in *Lecture Notes in Computer Science:Languages and Compilers for Parallel Computing*, Vol. 1366, Z. Li, P. Yew, S. Chatterjee, C. Huang, P. Sadayappan, and D. Sehr, Ed. Heidelberg, Germany: Springer Berlin, 1998, pp. 277-291.

[38] H. Dietz, T. Mattox, “Development of an Aggregate Function Message Passing Interface (MPI 2.0) Library,” *School of Electrical and Computer Engineering Annual Research Summary: July 1, 1998 – June 30, 1999*. Purdue, Indiana, USA: Purdue University, 1999. Available: https://engineering.purdue.edu/ECE/Research/ARS/ARS99/PART_I/Section4/4_16.whtml. [Accessed: August 5, 2009].

[39] H. Dietz, B. Dieter, R. Fisher, and K. Chang, “Floating-Point Computation with Just Enough Accuracy,” in *Lecture Notes in Computer Science: Computational Science – ICCS 2006*, Vol. 3991, V. Alexandrov, G. Albada, P. Sloot, and J. Dongarra, Ed. Heidelberg, Germany: Springer Berlin, April 2006, pp. 226-233.

[40] H. Dietz, “Linux Parallel Processing HOWTO,” The Linux Documentation Project, June 28, 2004. [Online]. Available: http://tldp.org/HOWTO/Parallel-Processing-HOWTO.html. [Accessed: August 5, 2009].

[41] M. Quinn, *Parallel Computing Theory And Practice*, 2nd ed. New York, USA: McGraw Hill, 1994.

[42] R. Fisher, and H. Dietz, “Compiling for SIMD Within a Register,” in *Proceedings of the 11th international Workshop on Languages and Compilers For Parallel Computing*, Vol. 1656, S. Chatterjee, J. Prins, L. Carter, J. Ferrante, Z. Li, D. C. Sehr, and P. Yew, Ed. Chapel Hill, NC, USA: Springer-Verlag, 1998, pp. 290-304.

Vita

Author's Name – Bobby Dalton Young

Birthplace – Memphis, Tennessee

Birthdate – October 10, 1983

Education

Bachelor of Science in Electrical Engineering

University of Kentucky

May – 2007

Research Experience

University of Kentucky

Lexington, KY

5/06 – 8/07, 1/08 – present

Graduate Research Assistant

Honors, Awards, and Activities

- Robert L. Cosgriff Award, University of Kentucky College of Engineering, 2006, University of Kentucky.
- Otis A. Singletary Scholarship, 2002 – 2006, University of Kentucky.
- Exhibitor, IEEE ACM Supercomputing Conference, 2006, 2007, 2008.

Society Memberships

- Eta Kappa Nu
- IEEE Student Member