ABSTRACT

JI, FENG. Runtime Support toward Transparent Memory Access in GPU-accelerated Heterogeneous Systems. (Under the direction of Xiaosong Ma.)

GPU has become a popular parallel accelerator in modern heterogeneous systems for its great parallelism and superior energy efficiency. However, it also extremely complicates programming the memory system in such heterogeneous systems, due to the non-continuous memory spaces on CPU and GPU, and a two-level memory hierarchy on a GPU itself. The complexity of this memory system is now fully exposed to programmers, who must manually move data to each position in the memory for correctness, and must consider data layout and locality to adapt to data access patterns in multi-threaded parallel codes for performance.

In this Ph.D. thesis study, we approach the above problem through providing runtime system support that aims at both easier programming and better performance. Specifically, we show two such runtime system software approaches, either within the scope of a programming model or as a general system software.

With the first approach, we focus on two popular programming models, MapReduce and MPI. For GPU-based MapReduce, we provide a transparent GPU memory hierarchy for MapReduce developers and realize performance improvement through buffering data in GPU’s shared memory, a small on-chip scratch-pad memory. On a system powered by an Nvidia GTX 280 GPU, our MapReduce outperforms a previous shared-memory-oblivious MapReduce, with a prominent Map phase speedup of 2.67x on average. For MPI, we extend its interface to enable the usage of GPU memory buffers directly in communications, and optimize such GPU-involved MPI intra-node communication, through pipelining CPU-GPU data movement with inter-process communication, and GPU DMA-assisted data movement. Comparing to manually mixing GPU data movement with MPI communication, on a multi-core system equipped with three Nvidia Tesla Fermi GPUs, we show up to 2x bandwidth speedup through pipelining and an average 4.3% improvement to the total execution time of a halo exchange benchmark; our DMA-assisted intranode data communication further shows up to 1.4x bandwidth speedup between near GPUs, and a further 4.7% improvement on the benchmark.

With the second approach, we present the design of Region-based Software Virtual Memory (RSVM), a software virtual memory running on both CPU and GPU in an asynchronous and cooperative way. In addition to automatic GPU memory management and GPU-CPU data transfer, RSVM offers two novel features: 1) GPU kernel-issued on-demand data fetching from the host into the GPU memory, and 2) intra-kernel transparent GPU memory swapping into the main memory. Our study reveals important insights on the challenges and opportunities of building unified virtual memory systems for heterogeneous computing. Experimental results
on real GPU benchmarks demonstrate that, though it incurs a small overhead, RSVM can transparently scale GPU kernels to large problem sizes exceeding the device memory size limit. It allows developers to write the same code for different problem sizes and further to optimize on data layout definition accordingly. Our evaluation also identifies missing GPU architecture features for better system software efficiency.
Runtime Support toward Transparent Memory Access
in GPU-accelerated Heterogeneous Systems

by
Feng Ji

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2013

APPROVED BY:

______________________  ______________________
Frank Mueller             Yan Solihin

______________________  ______________________
Huiyang Zhou             Xiaosong Ma
Chair of Advisory Committee
Feng was born in Nanjing, Jiangsu, CHINA, where he spent his childhood. He attended Nanjing Foreign Language School from 1996 to 2002. After that he went to Zhejiang University in Hangzhou, Zhejiang, CHINA. He spent six years there, studying computer science, and got his Bachelor and Master degrees in 2006 and 2008 respectively. He then moved to Raleigh, NC, USA to start his Ph.D. study until this thesis was finished in 2013.
ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Xiaosong Ma, for her lasting patience and guidance on my PhD study for the past five years. Dr. Frank Mueller, Dr. Yan Solihin, and Dr. Huiyang Zhou have been in my PhD committee since Apr 2012. I appreciate not only their time and effort but also their encouragement and suggestions in criticizing my work. My fellow members from PALM group have always been on my side, and have given me various types of help day to day.

For the work completed in this PhD study, I have collaborated with Ashwin Aji, from the SYNERGY group of Virginia Tech led by Dr. Wu Feng, the MPICH group from Argonne National Lab, primarily Dr. Pavan Balaji, Dr. Jim Dinan, and Dr. Darius Buntinas, as well as Dr. Heshan Lin, when he is Virginia Tech. I have received tremendous help from them when we work on projects and papers together. In addition their working styles and professional attitudes have been and will still be my examples.

My work is supported by my advisor’s funding grants from NSF and ORNL. I would take the chance to thank the agencies.

Last but not the least, I want to thank my family, my wife Ren Zhimin and my parents Ji Shifu and Shi Xiaofeng. Their understanding and support are always there in easy or hard time, which makes feel that I can finish this work.
# TABLE OF CONTENTS

LIST OF TABLES .................................................. vii

LIST OF FIGURES ................................................ viii

Chapter 1 Introduction ............................................. 1
  1.1 GPU-based MapReduce ........................................ 2
  1.2 GPU-aware MPI .............................................. 3
  1.3 RSVM: a Region-based Software Virtual Memory for GPU .... 4
  1.4 Contributions ................................................ 4
  1.5 Thesis Organization ......................................... 5

Chapter 2 Background and Related Work ......................... 6
  2.1 GPU Computing .............................................. 6
  2.2 GPU architecture .......................................... 6
  2.3 Heterogeneous System Architecture ......................... 8
  2.4 CUDA and NVIDIA GPUs .................................... 9
  2.5 Related work ............................................... 9
  2.5.1 MapReduce ............................................. 9
  2.5.2 GPU-aware MPI ....................................... 11
  2.5.3 GPU memory managment ............................... 11

Chapter 3 Using Shared Memory to Accelerate MapReduce on Graphics Processing Units .... 13
  3.1 Introduction ............................................... 13
  3.2 A Background of MapReduce ............................... 14
  3.3 Exploiting Shared Memory for MapReduce ................. 15
  3.3.1 Rationale ............................................. 15
  3.3.2 Shared Memory Layout ............................... 17
  3.3.3 Intra-block Thread Partitioning and Synchronization ... 18
  3.3.4 Result Collection and Overflow Handling ............. 20
  3.4 Evaluation ............................................... 21
  3.4.1 Testbed .............................................. 21
  3.4.2 Workloads .......................................... 21
  3.4.3 Memory Usage Modes Evaluated ....................... 22
  3.4.4 Map Results ....................................... 24
  3.4.5 Reduce Results .................................... 26
  3.4.6 Comparison with Mars ............................... 27
  3.4.7 Synchronization on GPU ............................. 30
  3.5 Concluding remarks and Future Work .................... 31

Chapter 4 Efficient Intranode Communication in GPU-Accelerated Systems ........ 32
  4.1 Introduction ............................................... 32
  4.2 Background ............................................... 33
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.1</td>
<td>The Message Passing Interface</td>
<td>33</td>
</tr>
<tr>
<td>4.2.2</td>
<td>MPICH2 Intranode Communication Architecture</td>
<td>33</td>
</tr>
<tr>
<td>4.3</td>
<td>Challenges in CUDA+MPI Programming</td>
<td>34</td>
</tr>
<tr>
<td>4.4</td>
<td>Design</td>
<td>35</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Integrating GPU-Awareness in the MPI Interface</td>
<td>36</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Eliminating Unnecessary Copies</td>
<td>36</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Increasing Throughput with Pipelining</td>
<td>37</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Efficient Host-Device Data Movement</td>
<td>39</td>
</tr>
<tr>
<td>4.5</td>
<td>Evaluation</td>
<td>43</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Exploration of Buffering and Message Queue Parameters</td>
<td>43</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Performance Comparison with Manual CUDA+MPI</td>
<td>45</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Evaluation with 2-D Stencil Benchmark</td>
<td>46</td>
</tr>
<tr>
<td>4.6</td>
<td>Concluding Remarks</td>
<td>47</td>
</tr>
</tbody>
</table>

Chapter 5 DMA-Assisted, Intranode Communication in GPU Accelerated Systems | 49
---|
| 5.1 | Introduction | 49 |
| 5.2 | Background | 50 |
| 5.2.1 | MPI and MPICH | 50 |
| 5.3 | Design | 51 |
| 5.3.1 | DMA-assisted Intranode Communication | 51 |
| 5.3.2 | LMT Protocol for Peer-GPU Communication | 52 |
| 5.3.3 | Intranode MPI Communication between Host and Device | 54 |
| 5.3.4 | Efficient Memory Handle Management | 54 |
| 5.4 | Evaluation | 55 |
| 5.4.1 | Experimental Setup | 55 |
| 5.4.2 | DMA-assisted Intranode MPI Communication between GPU Devices | 56 |
| 5.4.3 | DMA-assisted Intranode MPI Communication between Host and Device | 58 |
| 5.4.4 | Application Evaluation: Stencil2D | 59 |
| 5.5 | Concluding Remarks | 61 |

Chapter 6 RSVM: a Region-based Software Virtual Memory for GPU | 62
---|
<p>| 6.1 | Introduction | 62 |
| 6.2 | Background | 63 |
| 6.3 | Region-based CPU-GPU memory management | 64 |
| 6.3.1 | Region as Memory Management Unit | 64 |
| 6.3.2 | Benefit of Region-based Approach | 66 |
| 6.4 | RSVM Design | 66 |
| 6.4.1 | Region API | 68 |
| 6.4.2 | Region States | 69 |
| 6.4.3 | Region Management | 70 |
| 6.4.4 | CPU-GPU Region Consistency | 71 |
| 6.4.5 | Enabling Swap for GPU Memory | 71 |
| 6.4.6 | A Software TLB for Region Mapping on GPU | 72 |
| 6.5 | Implementation | 73 |</p>
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.5.1 GPU Callback</td>
<td>73</td>
</tr>
<tr>
<td>6.5.2 Lower-level Memory Allocator on GPU</td>
<td>73</td>
</tr>
<tr>
<td>6.5.3 Limitation of current implementation</td>
<td>74</td>
</tr>
<tr>
<td>6.6 Evaluation</td>
<td>74</td>
</tr>
<tr>
<td>6.6.1 Benchmark</td>
<td>74</td>
</tr>
<tr>
<td>6.6.2 Matrix multiplication with increasing problem size</td>
<td>75</td>
</tr>
<tr>
<td>6.6.3 Breadth-first search</td>
<td>76</td>
</tr>
<tr>
<td>6.6.4 Discussion</td>
<td>79</td>
</tr>
<tr>
<td>6.7 Concluding Remarks</td>
<td>79</td>
</tr>
</tbody>
</table>

Chapter 7 Conclusion ................................................. 80

References ............................................................. 81
LIST OF TABLES

Table 3.1 Workloads and problem sizes ........................................... 22
Table 3.2 MapReduce workloads characteristics, which shows the statistics collected
with the ”large” problem size. (1. All keys and values represent mean/standard
deviation of record sizes. 2. Ratios are input:output record count ratios. 3.
'-' means there is no reduce phase.) .................................................. 22
Table 5.1 Keeneland and Magellan Architecture and GPU Topology ........... 56
Table 6.1 Region APIs and region collection utility functions ..................... 69
Table 6.2 RSVM vs native in benchmarks that fit in the device memory (BackProp
has two kernels) ................................................................................. 74
Table 6.3 Benchmark graphs (n and m in 10^6). Small (upper, fit GPU memory limit)
and large (lower, exceed GPU memory limit). “kernel”: launching config-
uration as “number of blocks/block size” .......................................... 77
LIST OF FIGURES

Figure 2.1  GPU Architecture ................................................. 7
Figure 2.2  GPU Computing System Architecture .......................... 8
Figure 2.3  Current heterogeneous system model .............................. 9

Figure 3.1  Ideal Workflow of Map/Reduce Kernel empowered with Shared Memory.
Step 1: Stage in; Step 2: Computing; Step 3: Staging out. (Solid line: record boundary; dotted line: bytes) ........................................ 16
Figure 3.2  Workflow of MapReduce on GPU using shared memory .................. 17
Figure 3.3  Shared Memory Layout ............................................. 18
Figure 3.4  Map/Reduce kernels execution time in the large problem size ........... 24
Figure 3.5  Execution time composition of Mars and our framework in different memory usage modes at different problem sizes ......................... 28
Figure 3.6  Map/Reduce Phase Speedup over Mars ............................. 29
Figure 3.7  Kernel speedup by using GPU synchronization with yielding over non-yielding ..................................................... 30

Figure 4.1  Intranode latency and bandwidth between two MPI processes, with different source/destination buffer location choices. When a GPU device buffer is used, its associated host buffer is pinned. .................................. 35
Figure 4.2  Memory copies in (a) CUDA+MPI and (b) MPIGPU. ................. 37
Figure 4.3  LMT shared copy buffer ............................................. 38
Figure 4.4  Bandwidth of different GPU memory copy methods between host and GPU device. Adapted from CUDA SDK bandwidth test [11]. Host buffer is pinned always. ...................................................... 42
Figure 4.5  Bandwidth of different memory copy methods on the host side. Adapted from bw_mem bcopy from lmbench [46]. (MPIU_Memcpy is turned on for x86 but not x86-64 by default.) .................................................. 43
Figure 4.6  Bandwidth of two intranode MPI processes, with different shared buffer unit sizes, using messages at 4 MB, 1 MB, and 256 KB. Source and destination buffers both in GPU memory (“D2D”), both in main memory (“H2H”), or source in main memory and destination in GPU memory (“H2D”); the result of “D2H” is similar to “H2D.” ................................. 44
Figure 4.7  Latency/bandwidth of two intranode MPI processes, source and destination buffers both in GPU memory, using different message queue element sizes. The optimal shared buffer is shown for comparison. The results of “Device-to-Host” and “Host-to-Device” are similar. ................................. 45
Figure 4.8  Comparing MPIGPU with CUDA+MPI manual, on latency/bandwidth of two intranode MPI processes. Source and destination buffers both in GPU memory (“D2D”), source in main memory and destination in GPU memory (“H2D”), and source in GPU memory and destination in host memory (“D2H”). ................................................................. 46
Figure 4.9 Stencil2D performance improvement of MPIGPU over CUDA+MPI. The problem size is fixed when the number of processes are varied.

Figure 5.1 LMT Protocol for GPU Peer-to-Peer Communication

Figure 5.2 MPI intranode latency/bandwidth comparison between using DMA-assisted intranode communication between peer-GPUs – in different LMT modes (“coop”, “get”, “put”) – and using Nemesis’ shared memory protocol (“shm”). Near case on Keeneland.

Figure 5.3 MPI intranode latency/bandwidth comparison between using DMA-assisted intranode communication between peer-GPUs – in different LMT modes (“coop”, “get”, “put”) – and using Nemesis’ shared memory protocol (“shm”). Far case on Magellan.

Figure 5.4 MPI intranode latency/bandwidth comparison between using DMA-assisted peer-to-peer communication in different LMT modes (“coop”, “get”, “put”) and using shared memory (“shm”). Sharing GPU on Keeneland.

Figure 5.5 MPI intranode latency/bandwidth comparison between using DMA-assisted peer-to-peer communication with different DMA engine choice (“relay”, “attach”) and using shared memory (“shm”). When relay, two GPUs connected to the same chipset are used, on Keeneland.

Figure 5.6 Stencil2D performance improvement on Keeneland, comparing using DMA-assisted peer-to-peer communication against using shared memory buffer based communication, with varying number of MPI processes.

Figure 6.1 Dense matrix multiplication (A x B = C) GPU code example and execution behavior

Figure 6.2 Dense matrix multiplication (A x B = C) region programming example and running behavior

Figure 6.3 RSVM system architecture

Figure 6.4 An MSI-like state transition protocol

Figure 6.5 RSVM vs. native/manual matrix multiplication with increasing size (sum of input and output matrices)

Figure 6.6 RSVM vs. native BFS on small graphs

Figure 6.7 RSVM vs. manual/UVA BFS on large graphs
Chapter 1

Introduction

Graphics Processing Units (GPU), originally designed for graphics rendering, was later used in general purpose computing (GPGPU). The massively parallelism on GPUs made them a good fit for parallel computation. For the last several years, success stories across various domains have been reported on using GPUs to accelerate computations. Meanwhile, GPU designers are bringing more full-fledged computing functions and programming environment support to GPGPU program developers. Today GPU acceleration appears in both personalized computers and supercomputer-scale data centers.

However, as a co-processor, a GPU device brings complexity into a system architecture, i.e. a heterogeneous system architecture, which is equipped with both GPU devices and modern multi-core CPUs. The complexity has two sources: 1) CPUs and GPUs have different instruction sets (ISA), and hence different programming languages and non-compatible codes; 2) GPUs bring a non-continuous and hierarchical memory system in the heterogeneous system. While the former problem relies on parallelization compilation techniques, which still follows traditional parallel computing experience to a certain extent, the latter is a new problem and has not been fully addressed so far.

The memory system of a heterogeneous system has the following features. First, the introduction of a separate GPU memory space challenges the assumption of a single address space presented by the virtual memory system, bringing a non-continuous memory system. Moving data across the GPU memory and the CPU main memory is now a manual task of programmers. Second, GPU itself presents a two-level memory hierarchy: the large off-chip global memory and the small but fast on-chip shared memory. Moving data between them is a manual task as well. Besides, good performance of GPU codes often require carefully designed data movement plans to place and organize data in different places of the memory hierarchy. The traditional view of a single memory space across CPU cores and transparent hardware cache has provided much help for easy programming. But in order to better use GPU-powered heterogeneous systems, today’s
developers must spend a substantial period of time writing code for the memory hierarchy.

To solve this problem, we propose to provide support to facilitate memory accesses in GPU-enable heterogeneous systems. As current heterogeneous systems lack architectural support in the hardware, we have designed a set of software runtime approaches. Our software runtimes fall into two categories: 1) a runtime that runs transparently behind a certain programing model and manages such memory accesses, and 2) a general system-level software, which is not limited to a programming model. In the first category, we have studied two popular programming models, MapReduce and MPI. In MapReduce, we hide the GPU’s on-board memory hierarchy by providing transparent input and output data buffering in GPU’s shared memory, providing performance benefit for running map and reduce routines. For MPI, we design and implement a GPU-aware MPI system, which enables programmers to directly use GPU memory buffers in MPI function calls. Meanwhile, we devise optimization techniques within the MPI system, including pipelining inter-process communication and data movement across CPU and GPU, and GPU DMA-assisted intranode data movement, which shows optimized latency and throughput over manually mixing MPI with GPU data movement commands. In the second category, we design and implement a software virtual memory across the CPU-side main memory and the GPU memory, called RSVM, which takes over data movement across two separated memory domains. It also provides novel features, including fine-grained on-demand data fetching across CPUs and GPUs and transparent swap for the GPU memory.

In the following sections, more details will be given for each system.

1.1 GPU-based MapReduce

MapReduce [23] was first proposed by Google researchers as a parallel computation paradigm, as well as a runtime system in their distributed computing environment [23]. Later this elastic model was exploited for heterogeneous CPU processors [54, 55] and GPU processors [28]. In particular, Mars [28], a state-of-the-art MapReduce library, has shown advantage in throughput over its CPU counterpart for certain workloads. Beside performance benefits, general-purpose MapReduce frameworks like Mars simplify GPU parallel programming. Programmers can focus on correctness of their Map/Reduce function implementation, since the underlying scheduling, fault-tolerance, and performance issues are handled by the framework.

However, designing an efficient MapReduce framework on GPUs is still challenging for two reasons. First, many MapReduce workloads have irregular memory access patterns, which prohibits their efficient execution on GPUs. Second, though shared memory has been shown as able to accelerate many workloads on GPU, it is still unknown how it can be leveraged for MapReduce.

Therefore we investigate data placement and transfer between shared memory and global
memory for MapReduce applications running on GPUs, and propose a systematic approach to utilizing GPU shared memory as a transparent staging area to buffer MapReduce input/output data.

We will cover the details of this work in Chapter 3.

1.2 GPU-aware MPI

The Message Passing Interface (MPI) [10] is the industry standard for parallel programming and is used on virtually every high performance computing system. Introduced in 1984, the MPI standard has evolved to complement changes in CPU architecture and high-performance networks. One of the current challenges faced by the MPI community is the evolution of this popular parallel programming model to interoperate with and exploit GPU accelerators.

Current implementations of MPI continue to assume that all communication buffers are located in the main memory. Hence, developers must explicitly move data between device and host memories in order to perform MPI operations. With current GPUs, this strategy results in high-latency data movement across the PCIe bus and through several temporary buffers, even when performing communication between processes located on the same node. To facilitate these transfers, developers today typically maintain duplicate buffers in host and GPU memory manually, an approach that not only introduces code complexity but also is wasteful of system resources. Furthermore, one of MPI’s strengths is its ability to hide the performance details of data movement in parallel computations. This capability is significantly diminished when manual GPU data movement is performed outside the MPI library. In an experiment conducted with a pair of MPI processes on the same physical machine, we find that the bandwidth achieved is only half the theoretical peak when manual data movement is used.

To address this challenge, we extend MPI with direct support for the GPU memory space for communication. With this extended interface, GPU memory buffers can be used directly in MPI communication routines. Meanwhile, we adopt two optimizations within MPI for intranode data movement across co-located processes.

First, we pipeline data movement in PCI-express bus and MPI inter-process data movement in its internal shared memory buffers. The details of this work is described in Chapter 4.

In spite of the performance improvement in the above pipelining optimization technique, host memory is still involved, even for GPU devices connected to the same chipset. Therefore, we utilize the GPU DMA engines to further accelerate intranode MPI communication to realize direct data movement between GPUs. This part is elaborated in Chapter 5.
1.3 RSVM: a Region-based Software Virtual Memory for GPU

In today’s heterogeneous systems, GPU device memory is not managed by the operating system’s virtual memory. One needs explicitly manage the GPU memory space and manually move data between it and the host memory. Optimizing data movement on the PCIe bus brings another factor of complexity. The manual programming overhead greatly increases programmers’ burden in writing GPU codes, limiting their productivity in application development.

Although the lack of CPU-GPU memory management has been recognized by the systems research community [27, 33, 57], a unified virtual memory solution remains unknown. Several recent projects to build compilers [32, 33, 51] and task scheduling runtimes [27, 57] share the need of managing CPU and GPU memory spaces.

However, as today’s GPU hardware provides no virtual memory support, long available in modern CPUs, existing compiler techniques rely on host-side static analysis to estimate a GPU task’s input and output data sets. Thus, they do not enable on-demand data fetching for the GPU task, i.e. transferring only the needed data across the PCIe bus at run time. For the same reason, existing runtime approaches either require non-transparent ways of specifying GPU kernels’ data demand upfront, e.g. using directed acyclic graphs [57], or achieve on-demand data fetching only from the host side [27].

Picturing a unified virtual memory that hides the existence of discrete CPU and GPU memory domains, we propose a Region-based Software Virtual Memory (RSVM), to explore the design space possibility and challenges. RSVM abstracts the separated main memory and GPU memory domains, and automates data movement across them. In addition, it realizes on-demand data fetching for both CPU and GPU sides, and achieves a transparent swap mechanism for GPU memory, which enables a GPU kernel to scale up its problem sizes beyond the GPU device memory limit. Through benchmark evaluation, we show a proof-of-concept implementation with existing GPU hardware support, which incurs a minimal overhead in most cases. Meanwhile, we identify missing architectural features on current GPUs for better system software efficiency.

This part is elaborated in Chapter 6.

1.4 Contributions

In summary, this thesis work has made the following contributions:

- Our work on MapReduce on GPUs shows the approach of using GPU shared memory to buffer the various-sized input and output records in MapReduce workloads. Performance improvement is achieved while the GPU on-board memory hierarchy is made transparent to MapReduce developers.
• Our work on extending MPI with GPU native support shows the possibility of evolving MPI programming model for GPU-enabled heterogeneous system memory hierarchies. Optimization opportunities are thus present, and we have shown two approaches by leveraging MPI’s internal shared memory buffers and GPU DMA engines for intranode communications.

• Our work on RSVM shows the possibility of building a software virtual memory, which has runtimes on both the CPU and the GPU. Based on regions as the basic memory unit, RSVM provides the benefit of a virtual memory to abstract non-continuous memory domains. Our evaluation on RSVM overhead also reveals our insight into future architectural support required by GPU systems.

1.5 Thesis Organization

In the next chapter, a background on GPU and GPU computing will be given for the whole study; the background for each specific part of the work will be covered within subsequent chapters. Related works will also be given there. In Chapter 3-6, each part of this study will be discussed in detail. Chapter 7 concludes this thesis.
Chapter 2

Background and Related Work

This chapter provides a general background on GPU computing, covering the architecture of a GPU device and heterogeneous systems, and CUDA, a popular GPU programming model for Nvidia GPUs. After that, the related work is surveyed and organized by each topic.

2.1 GPU Computing

Graphics processing units were originally designed for rendering workloads. In recent years, the highly parallel GPU architecture was found to be extremely useful as a computation accelerator across a broad range of high-performance computing workloads.

Current GPU devices fall into two broad categories: integrated and discrete. Integrated GPUs are built into the same hardware components as the host processor and share memory and other resources. Discrete GPUs, on the other hand, are distinct from the host processor and tend to utilize a distinct memory subsystem. The larger chip area, more powerful parallel processing units, and high-throughput dedicated memory on discrete GPUs provide great performance potential and have made discrete GPUs a preferred design in current high-performance computing (HPC) systems. Current discrete GPUs are expansion cards that connect to the host processor and memory through the PCIe bus. Thus, data in host and device memories must be explicitly copied by using special commands.

2.2 GPU architecture

The architecture of a modern GPU [49] is shown in Figure 2.1. A GPU device has dozens of cores with duplicate ALUs and a three-level memory hierarchy. Each core, called streaming multiprocessor (MP or SM), consists of a group of scalar processors (SPs), a large register file. Each MP has a private L1 cache and they all share an L2 cache. and A part of the per-SM L1
cache, called *shared memory*, is a scratch-pad memory area for transient data storage. Shared memory is small (typically in the KB range) but fast (latency within dozens of cycles), and is shared by threads co-running on an MP. *Global memory*, in contrast, is off-chip and has a much larger size (typically in the GB range) and longer latency (400 - 700 cycles). Every several (e.g., 3 in Figure 2.1) MPs are grouped as a *Texture Processing Cluster* (TPC), equipped with a texture fetch unit and a 6KB-8KB per MP 2-level set-associative read-only texture cache [49,72]. Texture cache is used for texture loading and only caches data read from the global memory space bound to texture buffers. Designed for streaming fetches, a hit in the texture cache does not decrease fetch latency, but reduces the global memory bandwidth demand. As a read-only cache, it does not ensure coherence with writes to the same global memory space bound to the texture buffer.

When a GPU kernel is issued, a large number of threads will be launched on multiple MPs in a device. These threads are grouped as *blocks* in a grid. The blocks are required to execute in any order, sequentially or in parallel, which requires them to be mutually independent. When a kernel starts, as many blocks are started as possible on available MPs; the rest wait for their completion to be scheduled. Every 32 threads inside a block form a *warp* and run in lockstep synchronously. A warp is the basic scheduling unit in GPU, with multiple co-running warps providing an MP the chance to hide the long latency of accessing global memory. When threads in the same warp follow different execution paths, *warp divergence* happens, where each path is executed in serial.

Programmers need to carefully optimize their memory access patterns to achieve good GPU performance. In particular, for global memory, accesses from a *half-warp* (the first or the second half of a warp) can be *coalesced* into one transaction if the visited addresses are within an
aligned 32/64/128-byte segment. Otherwise up to 16 transactions will be issued. With new GPU architectures, accesses from one entire warp can be coalesced.

2.3 Heterogeneous System Architecture

Figure 2.2 shows a sample heterogeneous system architecture, based on Keeneland cluster nodes [69]. In the figure, multiple GPU devices are each connected via a PCI-Express (PCIe) interconnect to a chipset (hub), which is connected to other chipsets and CPUs. Different technologies can be used for the interconnect connecting chipsets, CPUs and memory, for example, the Intel Quick Path Interconnect (QPI) or the AMD HyperTransport (HT).

The bandwidth of the PCIe bus is orders-of-magnitude smaller than that of the GPU device memory. Thus, host-device data movement can easily become a bottleneck and hence deserves careful optimization.

Current GPU system model: In heterogeneous systems today, a GPU serves as a co-processor (as shown in Figure 2.3). The CPU is responsible for managing GPU resources through a driver program. For GPU computing, GPGPU APIs are provided by the driver for commanding the device. An application running on the host side can asynchronously launch tasks on the GPU. The host can also submit a synchronized command, which blocks till all the preceding commands complete.

The GPU memory and main memory are connected by the PCIe bus, and no hardware-enforced cache coherence is provided between the two memory spaces today. A GPU can visit a part of the page-locked main memory by mapping the host-side memory address into its own MMU. CPUs cannot visit the GPU memory, but only CPUs can issue commands to move data across the two places. GPU programmers must manually manage the GPU memory and transfer data between the host and the device memory buffers.
2.4 CUDA and NVIDIA GPUs

The Compute Unified Device Architecture (CUDA) is one of the most popular general-purpose parallel programming models on GPU today and is designed primarily for NVIDIA GPUs [50]. Our work focuses on CUDA because of its importance to current HPC applications. In the CUDA model, data between host and device memories is transferred by using the cudaMemcpy command. The programmer annotates functions in the program source code to enable them to run on the GPU. When invoked, these functions are launched as GPU kernels.

GPUDirect [7] is an NVIDIA technology that enables direct peer-to-peer GPU data transmission, completely using GPU DMA engines without any host processor or memory intervention. In the past, when data needed to be moved between the device memories of two GPUs, it had to be “staged” in the host memory. With GPUDirect, the data can be transferred from one device directly to another. However, this feature is currently restricted to peer accessible devices, i.e., those that are connected to the same AMD HT chipset. Further, GPUDirect does not currently support Intel QPI.

Another technology only available in CUDA, cudaIpc [6], allows different processes to access the same buffer located in a GPU’s device memory. A process can export a memory handle referring to a device memory buffer to another process. This feature is useful for parallel applications with multiple processes running on the same node, such as MPI applications.

2.5 Related work

2.5.1 MapReduce

MapReduce [23] was first proposed as a parallel programming model for distributed parallel computing. In the past several years it has attracted much attention as an elastic programming model to develop large-scale parallel processing jobs, and has been studied in other parallel computing environments. Phoenix [55] proves MapReduce can be used to program shared memory.

![Figure 2.3: Current heterogeneous system model](image)
multi-core and multiprocessor systems. Merge [40] uses MapReduce as a high-level language to program heterogeneous systems with both CPU and accelerators. Similarly, Sequoia [26] borrows MapReduce and builds memory hierarchy awareness into language support for portable parallel programs. On specialized processors, De Kruijf et al. presents a MapReduce system on Cell BE processor [22], and Papagiannis et al. revamps the design to minimize the control overhead and avoid redundant memory copies [52]. CellMR [53, 54] builds MapReduce on Cell BE processors in a cluster setting, and shows the benefit of streaming data across cluster nodes and asymmetric cores. On GPUs, besides Mars [28], Jiang et al. and Ravi et al. propose generalized reduction as an alternative API to MapReduce on Multi-core CPU and GPU [35, 56]. Other work on GPU [13,18] have also applied the MapReduce model to their algorithms. Among the above, our GPU-based MapReduce work is most similar to Mars as a general-purpose MapReduce framework for GPUs, though we are the first to explore the active use of shared memory in the GPU MapReduce workflow.

Most recently, MapCG [30] was proposed to abstract programming on both CPU and GPU with the MapReduce model, whose advantage over Phoenix [55] (CPU only) and Mars [28] (GPU only) was shown. For the GPU side, the authors also addressed the problem of two-phase running in Mars by using a light-weight memory allocator. Yet they did not use shared memory aggressively and their performance gain over Mars is primarily from building a hash table in the Map phase and replacing sorting with hash table lookups, which can be leveraged in our MapReduce framework in the future.

**GPU runtime features** Aside from a rich body of work on using GPU for specific algorithms, recently more research interest is attracted to runtime and programming language support for GPU-enabled systems. Mars [28] and our MapReduce work fall into this category. Ma develops a translational framework for data mining algorithms on GPU [45]. Sundaram et al. and Satish et al. use domain templates expressed in graphs to attack the problem of total data size not fitting on GPU and minimizing data transfer between host and GPU [59, 68]. Our MapReduce framework complements them by addressing efficient use of the GPU shared memory.

In addition, Zhang et al. proposes to use idle CPU cycles for thread reference redirection and data layout transformation to avoid GPU threads divergence [76]. Targeting a different problem, our technique of thread roles partitioning shows some resemblance to their thread reference redirection; the difference is that we have all things done at the GPU side.

As for synchronization on GPU, Xiao et al. [73] studies 3 different inter-block communication mechanisms on GPU. Our GPU-based MapReduce uses intra-block synchronization within one thread block and is complementary to their work. About explicitly managed shared memory, Silberstein et al. [62] propose a software-managed cache layer for sum-product algorithms. Our design of shared memory management is similar, but is tailored for one dimensional data buffer
and the overflow handling problem.

### 2.5.2 GPU-aware MPI

Accelerators such as GPUs put a challenge faced together by the MPI community: can MPI further adapt to the heterogeneous systems with both high-performance multi-core CPUs and GPUs with rather different architectures. At present MPI users are still struggling with an MPI+GPU programming model, by manually mixing a specific GPU programming model, e.g. CUDA [6] or OpenCL [38], into MPI framework source code. Meanwhile, a few different research directions regarding this challenge is under investigation recently.

As one direction, researchers are proposing modifications to the evolving MPI model. Stuart et al. suggested several potential ways for extending the MPI standard to provide native support of these accelerators [66]. One significant extension attempt is to allow GPU threads to obtain MPI ranks and participate in MPI communications [67]. But, due to lacking network I/O functions on GPUs, CPU helper threads are needed, which complicates performance modeling and may introduced new overhead. However, we believe MPI standard modification to be promising and perhaps necessary in the long term.

Another different direction is to follow the current MPI model, and extend it with transparent system solutions [7, 63, 70, 71]. The large number of existing MPI programs can benefit from this direction. In this sense, our work in extending MPI falls into this category, and differentiate itself from the existing ones in studying intranode communications as opposed to internode GPU-to-GPU communications.

While traditionally MPI has been well known for its powerful capability for writing cluster-wise parallel jobs, addressing the intranode parallelism has become equally important for an MPI implementation since the multi-core era [15, 16, 43, 44, 48]. Our work also complements this list by studying intranode communication subsystems for GPU-accelerated heterogeneous systems.

### 2.5.3 GPU memory management

**Program non-continuous memory hierarchy** Plenty of work has been done to facilitate transparent data movement for accelerators, e.g. Cell BE processors [19], GPUs, and Larrabee processors [60], in specialized programming models, such as MapReduce and directed acyclic graph based algorithms [14, 24, 26, 41, 42]. Transparent software caching techniques on accelerators have been proposed for CELL BE and Larrabee [25, 58, 75], by utilizing their virtual memory support, which is missing on today’s GPUs [47]. RSVM demonstrates the potential of software virtual memory without requiring hardware or OS support on GPUs.
**Compiler assisted CPU-GPU communication** Compiler techniques have been developed to attain automatic CPU-GPU communications [27, 32, 33, 51], without requiring virtual memory exception support on the GPU. Static program analysis techniques are applied to GPU kernel code, to estimate GPU kernels’ input and output data set, and a host-side runtime mechanism is employed to check states and initiate data movement. However, the static analysis on GPU code is ineffective in dealing with dynamic data movement requirement, when data access is decided dynamically or when the working set exceeds GPU physical memory size limit. RSVM resolves these issues through a GPU-side runtime and callbacks. The dynamic memory footprint revealed by RSVM relieves compilers from conservative estimation of GPU data demand. Meanwhile, compilers can help RSVM in identifying regions, by analyzing locality in the GPU kernel code.

**OS support for GPGPU** Though GPUs have been used in modern computer systems for over a decade, system designers have just noticed the limited OS authority over the GPU resources, as they are turning into compute devices. Previously delegated to GPU device drivers, scheduling, data flow, and QoS management of GPU tasks can be incorporated into the OS kernel space, e.g., GDev [37], possibly through new OS abstractions, e.g. PTask [57]. While they are more concerned with the synergy of current GPU programming models with the OS, RSVM is novel in extending system-level management into GPU kernels, to achieve transparent swapping and fine-grained on-demand data movement.

**Distributed shared memory** RSVM is inspired by distributed shared memory systems in the past [39] and particularly by CRL [36], an all-software distributed shared memory. Similar ideas were first introduced in ADSM [27], which abstracts a pair of user-managed host-side and device-side buffers. RSVM, however, adopts a different design and provides new features such as GPU-side on-demand data fetching and transparent swap. Very recently, GPUs [61] provides POSIX-like file I/O APIs for GPU kernels. It uses a device-side runtime to implement a buffer cache on the GPU. Despite the resemblance in design, RSVM’s memory-level interface is more flexible in programming CPU-GPU data sharing and co-processing. Meanwhile, GPUs’ functionality can be implemented on top of RSVM, by binding a region’s host-side address with memory-mapped files.

**GPU virtual memory architectural evolution** Today’s GPUs lack hardware virtual memory exception support, prompting RSVM to adopt a software approach. At the same time, new architecture such as Heterogeneous System Architecture (HSA) [9] is emerging for on-chip, integrated GPUs. This avoids the manual memory management problem by employing a single memory for both CPUs and GPUs. While its programming benefit is appealing, the current practice of deploying different memory types for CPUs and GPUs will likely offer performance advantage to GPGPU workloads.
Chapter 3

Using Shared Memory to Accelerate MapReduce on Graphics Processing Units

3.1 Introduction

In this Chapter, we give details about our work on GPU-based MapReduce. Specifically, we propose to hide the GPU memory hierarchy in the MapReduce programming model. We investigate the data placement and transfer between shared and global memory for MapReduce applications running on GPUs. Our main contributions are as follows:

- We proposed a systematic approach to utilizing GPU shared memory as a transparent staging area to buffer MapReduce input/output data. To our best knowledge, this work is the first to explore the active use of shared memory in MapReduce execution on GPUs.

- We devised several mechanisms, including shared memory staging area management, thread-role partitioning, and intra-block thread synchronization, to handle the dynamic memory management issues, especially for output overflow handling in the staging area.

- We implemented the above techniques in a general-purpose MapReduce framework on GPUs, which needs no two-pass execution and makes no assumption on input/output data sizes, size variations, or record counts.

- We evaluated our framework extensively with five common MapReduce workloads, and compared our framework with Mars. The results indicate that it is worthwhile to exploit the small shared memory. Especially, output staging does bring a significant performance improvement. We also conducted comparison among input staging, texture buffering, and
simply using global memory, and showed that our staging input mechanism performs better than or comparably to using the texture cache in most cases.

3.2 A Background of MapReduce

MapReduce [23] was proposed as a programming model for data processing jobs on large-scale clusters. With programs written in a functional style, the MapReduce runtime takes care of data partitioning, task scheduling, inter-machine communication, and fault tolerance. The MapReduce workflow consists of three phases: Map, Shuffle, and Reduce. Map and Reduce are two routines written by user programmers, who also provide specifications of input, intermediate results, and final output. When a job starts, the framework spawns a number of Map tasks, each working on a subset of input records and generating intermediate results. In the shuffle phase, the framework aggregates the intermediate results into key sets, each with a distinct key. Finally, the framework spawns a Reduce task per key set and generates final output records. Input, intermediate, and output are all in the format of \(<key, value>\) pairs, though the definition of key and value for each is the choice of programmers.

The data parallel computation model of MapReduce fits GPUs well. However, compared to traditional applications running on GPUs, MapReduce workloads with variable sized input or output create unique challenges to be tackled on GPUs. This is due to the irregular memory accesses in MapReduce workloads and the lack of efficient dynamic memory management mechanisms, so far, in either shared memory or global memory. Mars [28], a state-of-the-art GPU MapReduce framework, uses a two-pass execution mechanism, requiring extra MapCount and ReduceCount phases before Map and Reduce phases respectively, to deal with memory management for output. The first pass, MapCount or ReduceCount, is only used to compute the output sizes of each task. At the end of this pass, a prefix summing operation is executed across all threads with output size values in order to find their own starting output address. This way, output data can be written subsequently in the second (“real”) pass, to memory address ranges set appropriately by the prefix summing step. Due to the lack of atomic operations on previous generations of GPUs, such a two-pass mechanism turns out to be a practical approach for variable sized output. Part of our contribution in this work is to remove the cost of such two-pass execution, by leveraging the atomic global memory access method and our shared memory management scheme.

Second, though shared memory has been shown as able to accelerate many workloads on GPU, it is still unknown how it can be leveraged for MapReduce. Among related studies, Archuleta et al. first investigated the shared memory usage for input in temporal data mining [13], a MapReduce-like workload. However, general MapReduce workloads can be different in their varying record sizes, and using shared memory for output has not been examined. In
addition, Catanzaro et al. [18] examined on GPU a machine learning MapReduce workload. Both workloads mentioned above have output size/count known a priori. The aforementioned Mars framework [28] has a minimal use of shared memory and leaves its usage as developers’ call.

3.3 Exploiting Shared Memory for MapReduce

3.3.1 Rationale

With the advent of atomic operations on GPU, having an appendable output buffer in global memory becomes realizable. However, due to the large number of concurrently running GPU threads, a global lock or a critical section for accessing appendable output buffers can become a bottleneck due to severe competition (we will show this in the next section across a few workloads). Therefore we propose using shared memory as input and output buffer for MapReduce workloads to relieve this contention. The benefit of using shared memory is listed as follows.

- Coalesced global memory transactions: when data are moved by our MapReduce framework between global memory and shared memory, GPU threads can follow strictly the coalesced global memory access pattern to attain the best memory bandwidth. MapReduce programmers need not manually coalesce memory accesses when providing Map and Reduce routines, since data is already buffered in shared memory in a transparent manner.

- Fast shared memory access: Map and Reduce routines can enjoy the short latency of shared memory. When access locality exists in the computation, fewer global memory requests will be generated.

- Reduced contention on result collection: having a temporary output buffer in shared memory forms a hierarchical result collection mechanism, which can remove the severe contention caused by single-level global memory result collection from a large number of threads.

An ideal workflow of MapReduce on GPU incorporating the use of shared memory, as shown in Figure 3.1, consists of 3 steps:

**Staging in** All threads in a block cooperate on moving portions of input data from global memory to an input area in shared memory. In this step, without knowledge of records, these threads only see the data as a contiguous range of bytes in global memory. Using the start and end addresses of the range, neighboring threads from a warp always move contiguous primitive-type data to allow coalesced global memory accesses. Staging in action is repeated until the shared memory input area is full.
Figure 3.1: Ideal Workflow of Map/Reduce Kernel empowered with Shared Memory. Step 1: Stage in; Step 2: Computing; Step 3: Staging out. (Solid line: record boundary; dotted line: bytes)

**Computing** Each thread picks a record at a time, locating its address according to the directory indices (also staged into shared memory), and starts the user-provided Map/Reduce function on each record. Results are written into the output area, also in shared memory. The three input records shown in Figure 3.1 have 2, 3, and 2 bytes respectively, and generate three output records with sizes of 1, 3, and 2 bytes respectively.

**Staging out** All the threads in the block cooperate on flushing output data from shared memory to global memory. Again, global memory accesses are coalesced.

The above three steps are repeated until all input records are processed. Here we assume global memory is large enough to hold the output, considering today’s global memory size typically at the GB level. Otherwise, batched processing is again possible at another level and it is possible to overlap GPU kernel execution with host-device data transfer.

However, as mentioned earlier, a unique challenge for using the small shared memory for input/output staging is to coordinate the dynamic space usage. In particular, considering the lack of powerful inter-thread communication/synchronization schemes, as well as that MapReduce result sizes are often not known a priori, output overflow handling needs to be addressed carefully.

The complete view of our MapReduce framework on GPU to use shared memory for buffering input and output data is similar to the ideal workflow with 3 steps. Yet because of the possible overflow in the staging area, now it may have multiple computing and staging out steps per input iteration as shown in Figure 3.2. As an example, imagine a Map kernel launching with 256 threads per block, and in an input iteration, 118 input records are staged into the input area in the shared memory belonging to block 0. Then the first 118 threads will each pick one input record to start the user-provided Map function. During this process, these threads keep emitting intermediate results in the output staging area and may find that there is no space
left. At that moment, all 256 threads will flush the collected results in the output area to an output buffer in global memory. After the flushing is done, the 118 threads will resume the Map computation until another overflow or the final flushing at the end of this iteration. The detailed mechanism of our framework will be given in the following sections.

### 3.3.2 Shared Memory Layout

First, we partition shared memory into an input and an output area. Figure 3.3(a) illustrate the shared memory layout in our design. The size ratio between the input and output areas is a parameter dependent on workloads. There is a tradeoff in that a larger input area increases computation concurrency by allowing more threads to work simultaneously due to more resident input records, while a larger output area leads to fewer overflows and less overflow handling overhead. One can roughly estimate the input and output ratio of a specific workload to decide the parameter value. In our future work, we plan to make this configuration automatic and adaptive.

In the shared memory input area, four separate buffers are allocated to store keys, values, key indices, and value indices. These buffers are mapped to contiguous segments in their corresponding data buffers in the global memory, so input staging benefits from coalesced memory accesses. A separate small working area is allocated to each thread, for the storage of temporary variables used in Map/Reduce computation. All these input buffers are statically managed.

![Workflow of MapReduce on GPU using shared memory](image-url)
before the computing step starts.

The output area, or result collection area (Figure 3.3(b)), is managed dynamically. We define a *warp result* as the collection of results (either intermediate results after Map or final results after Reduce) generated by the threads of one warp in one result record generation - threads in a warp run in lockstep, which makes their results generation occur at the same time. Output in MapReduce has the same format with input and contains two types of data: 1) size-predictable, structured key indices and value indices; and 2) size-unpredictable, unstructured key and value pairs. To efficiently use the shared memory space, we use a double-ended stack to store the size-predictable structured data from the left end and the size-unpredictable unstructured data from the right end. For example, in Figure 3.3(b), two warp results are shown. Each warp result has two arrays of pointers, *i.e.* key indices and value indices, on the left, among which only the first key index and value index are drawn here. Keys and values of the same warp result are stored contiguously on the right. Compared to using four separate buffers as in input area, which easily causes internal fragmentation, this design will only see overflows when the total remaining output space is smaller than a newly generated warp result. Unlike in the input area, the four types of data across all results are not stored all contiguously here. However, when flushing a warp result back to global memory, coalesced global memory writing is still attainable within the scope of one warp result.

### 3.3.3 Intra-block Thread Partitioning and Synchronization

GPU supports a great number of threads per block. However, the concurrency in computation is limited by the input area size and the input record granularity. Due to the small size of shared memory, the number of input records staged in shared memory can be smaller than the thread block size. We can let the otherwise idle threads in a block assist in data staging.

In our framework, threads within a block are partitioned into *compute threads*, which carry
out Map/Reduce computation, and helper threads, which remain idle during computation but cooperatively handle result overflows. To avoid warp divergence, we divide them between warps, so that a warp is either a compute warp or a helper warp. As the concurrency may not be a multiple of the warp size, we increase the number of compute threads to the nearest multiple of the warp size - the last few compute threads of the last compute warp may not get an input record.

It is not too hard to enable a producer-consumer type of synchronization between the compute and helper threads with multi-threading programs on a CPU. On GPUs, however, there lack lower-level primitives for inter-thread coordination. Essentially, though there is a block-wise barrier (\texttt{syncthreads()}), no wait-signal type synchronization is available. Note that when threads enter the computation phase, the compute and helper threads branch into different execution paths. This makes the block-wise barrier inapplicable as its occurrence in branched paths leads to a hang or undefined result [49].

To solve this problem, we designed a software-based wait-signal primitive for synchronizations between the compute and helper threads. The condition is implemented with a set of flag variables in shared memory, each representing the status of one warp, considering the synchronous nature of its threads. We define wait group as the group of warps that wait on a condition, and signal group as the group of warps that signal on the condition.

Initially, all wait group and signal group flags are cleared. In a wait operation, the wait group threads poll at the signal flags until they see them all raised by signal group threads. Then the wait group warps change their own flags notifying the signal group that the signal is seen. When the signal group threads find all wait group warps in the “seen” state, they leave the signal routine and reset their flags. After setting their own “seen” state, the wait group threads check their peers’ states. The wait group warp who finds all its peers in the “seen” state, i.e., the last “seen” warp, resets all wait flags after seeing all signal flags clear, and leaves the wait routine; the other waiting group warps simply leave the wait routine when they find they are not the last one. Note that the size of the two groups must be known in advance. In our case, the partitioning of thread roles (compute vs. helper) is determined at the end of each input staging operation.

For efficiency we must let the helper threads enter a non-running (blocked or sleep) state lest they compete for cycles with the compute threads. But GPU threads do not have as many states exposed to software as CPU threads do. Therefore, we devised a mechanism to reduce the polling frequency. Since a global memory access has a large latency, the hardware scheduler swaps out the threads waiting for their operands ready, so that such global memory access latency can be hidden. With this we provide a yield operation for GPU threads by executing a dummy global memory read and write to a preallocated dummy global memory buffer. This will cause the calling threads to be scheduled off the MP.
Aside from synchronization, *memory consistency* is required: shared memory data updates by the signal group before signaling must be visible to the wait group when they see the raised flags. This requires *processor consistency*, a very weak consistency model which we have confirmed with Nvidia regarding their GPUs. Under this consistency model, all threads see writes (in shared memory in our case) from one thread in the same order as they were issued by that thread. Considering potential future architectural evolution to even weaker consistency models, a *threadfence* `block()` at the entrance of the signal routine ensures the consistency requirement of our synchronization and adds an ignorable (<1%) performance overhead.

With our synchronization mechanism, the workflow can now have multiple output staging steps per input iteration to handle output overflows, as illustrated in Figure 3.2. In the staging in and staging out steps, all threads participate in moving input or output between global memory and shared memory. When a compute step starts, only compute threads participate in the Map/Reduce computation, and the helper threads are “idle” (shown in dashed arrows), waiting on the *overflow* condition. When the helper threads see the signal of overflow, all threads start the staging output step. At the end of staging out, similar synchronization through another condition, *overflow-handled*, lets the compute threads continue with their execution and helper threads return to waiting.

### 3.3.4 Result Collection and Overflow Handling

We designed a hierarchical result collection mechanism, which happens both at the warp level and at the block level.

At the warp level, results emitted by threads are required to be temporarily in an addressable place before being passed into our framework as references: they can be saved either at their original locations in the input area (*e.g.* Word Count has an output key as part of its input) or in the preallocated working area (*e.g.* Matrix Multiplication saves a floating point number as output per thread). These results will be grouped into one warp result and stored in the shared memory output area. To enable parallel output record writing, a prefix summing process is conducted in the warp to get the relative starting address for each thread’s output. Since a warp of threads run in lockstep, no explicit synchronization is required among them during this process.

At the block level, the first thread in a warp takes two actions atomically: 1) append the structured data portion of a new warp result from the left end of the result collection buffer, and 2) reserve the total size of the unstructured portion of the warp result from the right end. When this is done, all threads from this warp can freely update its own pointer indices at the left end, and copy its own output key-value record from the addressable place to the reserved area at the right end. Again no synchronization is needed.
An overflow happens when there is not enough space for a new warp result. Overflow handling is started with the participation of all threads within a block. When staging-out starts, the first thread of the block calculates the total size requirement for all the warp results collected in shared memory, atomically increases the length of results stored in global memory, and reserves the appropriate global memory address range for all warp results. Our wait-signal primitive is used again here to coordinate the operations among threads. When the reservation is done, all warps iteratively flush all warp results to the reserved area in global memory, each responsible for dumping one warp result at a time.

3.4 Evaluation

3.4.1 Testbed

Our test bed is an AMD Athlon Dual-core 3800+ machine with 1GB main memory and an Nvidia GeForce GTX 280 card. GTX 280 card has 30 multiprocessor cores, each with 16384 32-bit registers, 16KB shared memory, and 1GB global memory. The OS is 64-bit Fedora 10 with Linux kernel 2.6.27, and Nvidia’s 64-bit Linux driver version 190.18.

3.4.2 Workloads

We select a set of five MapReduce workloads, shown in Table 3.1. They involve both variable and fixed sized records and have been used for evaluating existing frameworks such as Mars [28]. Our implementation follows the same algorithms as in Mars, with several different memory usage modes, to be detailed in Section 3.4.3.

Among the workloads, **Word Count (WC)** goes through a set of documents and calculates the number of occurrences for every distinct word. Each Map task takes a part of the input and emits a \(<\text{word}, 1>\) pair for each word it sees. Each Reduce task takes one distinct key (word) and sums all the values sharing the same key into an output key-value pair. **Matrix Multiplication (MM)** calculates the product of two input matrices. Each Map task takes one row and one column from the two input matrices, respectively, and calculates the value of one element in the result matrix. No Reduce phase is needed for MM. **String Match (SM)** searches for a given keyword in a document. Each Map task takes a line and searches for the keyword. If a keyword is found, the line is emitted as a result. No Reduce phase is needed for SM. **Inverted Index (II)** goes through a set of html files and finds all URL links. Each Map task takes one part of the input, and searches for a link. Whenever it finds one, it emits the link as well as the link’s position in the document. Again no Reduce phase is needed for II. **KMeans (KM)** groups a set of vectors into \(K\) clusters according to their distance in the space. Each Map task takes one vector and calculates its distance to \(K\) centroid vectors of existing
### Table 3.1: Workloads and problem sizes

<table>
<thead>
<tr>
<th>Workload</th>
<th>Problem Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word Count (WC)</td>
<td>16MB / 32MB / 64MB</td>
</tr>
<tr>
<td>Matrix Multiplication (MM)</td>
<td>512x512 / 1024x1024 / 2048x2048</td>
</tr>
<tr>
<td>String Match (SM)</td>
<td>16MB / 32MB / 64MB</td>
</tr>
<tr>
<td>Inverted Indexing (II)</td>
<td>16MB / 32MB / 64MB</td>
</tr>
<tr>
<td>Kmeans (KM)</td>
<td>4MB / 16MB / 64MB</td>
</tr>
</tbody>
</table>

clusters, and then emits as an intermediate result the id of the nearest cluster and the vector itself. Each Reduce task takes one cluster, and computes its new centroid.

For all of our workload results, we tested with three problem sizes (see Table 3.1), which are similar to those used in previous systems such as Mars. Table 3.2 summarizes the characteristics of these workloads, in terms of the mean and variance of record sizes for input, intermediate, and output, as well as the input-to-output record count ratios of the Map and Reduce phases.

*E.g.*, WC’s Map phase takes a string of characters, including words and punctuations, as an input key record. Such an input key has on average 32.44 bytes (with a standard deviation of 2.59). In addition, there is a 4-byte integer record index number as an input value record. WC generates words as intermediate result keys, with an average size of 5.46 letters and a standard deviation of 2.53, as well as a value "1" for each word. The input-to-output record count ratio means that there are 4.98 words per input record on average.

#### 3.4.3 Memory Usage Modes Evaluated

We examine the effectiveness of using shared memory in different components of MapReduce computation, by evaluating the following combinations of memory usage options:

- **SIO**: staging both input and output in shared memory, using the shared memory layout and data handling approach described in Section 3.3.

Table 3.2: MapReduce workloads characteristics, which shows the statistics collected with the "large" problem size. (1. All keys and values represent *mean/standard deviation* of record sizes. 2. Ratios are *input:output* record count ratios. 3. '-' means there is no reduce phase.)

<table>
<thead>
<tr>
<th>Workload</th>
<th>Input</th>
<th>Map</th>
<th>Interim</th>
<th>Reduce</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Key</td>
<td>Value</td>
<td>Ratio</td>
<td>Key</td>
<td>Value</td>
</tr>
<tr>
<td>WC</td>
<td>32.44</td>
<td>2.59</td>
<td>4/0</td>
<td>1:4.98</td>
<td>5.46/2.53</td>
</tr>
<tr>
<td>MM</td>
<td>8192/0</td>
<td>8192/0</td>
<td>1:1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SM</td>
<td>44.52/2.68</td>
<td>4/0</td>
<td>3.83:1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>II</td>
<td>8/0</td>
<td>63.9/123.2</td>
<td>7.94:1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>KM</td>
<td>0/0</td>
<td>32/0</td>
<td>1:1</td>
<td>4/0</td>
<td>32/0</td>
</tr>
</tbody>
</table>
• **SO**: staging only output in shared memory, but reading input directly from global memory. Intra-block thread synchronization is needed for output overflow handling.

• **SI**: staging input only in shared memory. Each thread is responsible for writing its own output record to global memory. In order to avoid extensive contention incurred by atomic accesses, only the first thread of each warp atomically increases the output size in global memory by the total size of all output records from its warp, calculated through in-warp prefix summing. This global memory range is then broadcast to other threads in the warp through shared memory.

• **G**: no staging. This mode is similar to the Mars scheme. However, it avoids the two-pass execution by using atomic operations in the same way as SI does.

The wait-signal intra-block synchronization primitive is only used when we stage output in shared memory (in SIO and SO). Also note that at least a warp (32 threads) are needed to enable intra-block synchronization. In SI and SIO, both with input staged, the number of compute threads running concurrently is subject to the thread block size (the number of threads within a block) and the input area size.

Side comparisons exist between using the texture cache and the SI or G modes. Staging output cannot be compared since the texture cache is readonly. Therefore, in addition to the four memory usage modes mentioned above, we have implemented a **GT** mode, which is similar to **G**, but binds the input in global memory to texture buffers. Please note that this mode requires a different version of user-provided Map/Reduction functions because their general memory accesses, i.e., pointer operations, in global memory or shared memory, must be replaced with explicit texture fetch directives.

Note that the MM Map kernel is different in that only the indices for a row/column vector can be staged into shared memory. Otherwise, the huge record (a row or column vector of a matrix) will reduce the concurrency to fewer than 8 threads, due to the small size of shared memory.

For Reduce phase, we implemented and evaluated two reduction strategies: **thread-level reduction (TR)** and **block-level reduction (BR)** [13]. With TR, each thread works on a distinct key set by running user-provided sequential code. With BR, a block of threads work on a distinct key set in parallel, by following a predefined reducing order, e.g. a tree-like order that reduces two children values at every node of the tree. Among existing MapReduce frameworks, TR is adopted by Mars and Hadoop [74], while BR is adopted in Catanzaro’s MapReduce framework on GPU [18]. Note that the TR implementation cannot stage input, as by definition it processes a complete key set at a time, which can be arbitrarily large. Map input, on the other hand, has reasonably small records to be processed one at a time, with the exception of workloads like...
MM. Besides, for BR kernels, we cannot implement the GT mode. This is because BR kernels update values in their original places, which becomes a problem as the texture cache coherence is not guaranteed with global memory writes in one kernel invocation.

![Figure 3.4: Map/Reduce kernels execution time in the large problem size](image)

### 3.4.4 Map Results

Figure 3.4(a)-3.4(e) show the results of Map kernels with different memory usage modes in the large problem sizes. The small and medium problem size results possess similar trends and are omitted due to the space limit.

From these experiments, we observe that utilizing GPU shared memory through input/output data staging does bring a significant improvement to the performance of Map kernels.
In particular, SIO, by combining input and output staging, delivers the best or close-to-best performance in most cases. Compared to G, SIO achieves an average 2.85x speedup, with a maximum of 7.5x.

However, the figures also reveal that the source of such performance benefit heavily depends on workload characteristics. For WC and SM, staging output makes more difference, while for II, staging input creates a larger impact. For KM, it seems that only by combining SO and SI can we receive a significant improvement.

With WC and SM, SO brings large performance gains (in most cases more than 2x speedup compared to G). This is due to their large number of Map output results. With frequent result generation, when output is not staged through shared memory, there is severe contention over a critical section involved in atomic global memory updates, for both output data and associated indices. Though we have reduced this contention by performing in-warp prefix summing, G and SI both suffers from the overhead and unscalability of this bottleneck. In particular, with these two workloads, both G and SI produce longer Map execution time when the number of threads per block increases, while SO and SIO benefit from the increased concurrency brought by more threads. WC, due to its large output-to-input ratio, receives more performance gain than SM from SO, while SM does benefit slightly from SI as it has more access locality when processing the input data.

II benefits significantly and solely from staging input. This can be attributed to several of its traits: 1) very large variance in input record sizes, 2) longer, more complex computation logic than WC and SM, with higher level of input access locality, and 3) a large ratio of input-to-output record count. The long and variant-sized input records and complex computation make staging input especially beneficial, by avoiding expensive global memory accesses. Yet as long as input is staged in shared memory, the computation concurrency is limited by the shared memory size and large-sized input records, making both SIO and SI performance quite flat with increased thread counts. SIO is slightly better than SI, between 128 to 256 of block size, due to the capability of avoiding the aforementioned output update bottleneck in global memory. Seeing the trend of better performance of G with increased thread count, we further experimented beyond 320 threads per block (not shown in the chart). The marginal decrease in G execution time gets smaller, losing to SIO by at least a factor of 1.5.

Though helpful in most cases, compared to SI, which never degrades performance, SO may bring new overhead and can even hurt the overall Map execution time sometimes (e.g., with II). Such overhead comes from two aspects: 1) Even with our synchronization mechanism based on dummy global memory accesses, the helper threads are still performing busy waiting during the computation phases. Such an overhead is more pronounced when the computation step is longer, as more busy waiting operations are conducted. 2) Early finished compute threads are busy waiting on their peers, to participate in potential overflow handling in the future. For the
former, the overhead can be reduced by shortening the computation phases. Note that staging input can effectively reduce the computation phase for most cases, so SIO is usually better than or equal to SO duration. For the latter, the variance in input record sizes is the primary reason for the uneven Map computation time (e.g. II-M).

KM possesses certain common characteristics with both WC and II, making it benefit most from the combination of SI and SO. On one hand, it has fixed sized input/output records, with an input/output size ratio close to 1. Due to the substantial output sizes, SO helps to relieve the problem of unscalable global memory accesses seen in G, especially with large thread blocks. On the other hand, it has strong access locality, by repeatedly accessing an input vector for each centroid, making staging input desirable. Most importantly, by leveraging such locality and shortening the computation phases with input staged into shared memory, the busy waiting overhead of staging output is reduced, making SIO a clear winner.

Finally, MM reads data anyway from global memory, bringing the four modes closer in performance. Further, as the workload is memory-bound, all four modes show similar unscalability with increased thread count per block. Small as the difference is, staging output can bring certain advantage here, when the thread block is large. With a block size of 64, the benefit of SIO and SO is offset by the fact that they have to leave a warp of 32 threads as helper threads, which halves the threads available for computation.

As for the effect of texture cache, at the cost of modification to Map function code, GT never loses to G. Comparing SI to GT, except for MM-M, where input cannot be completely staged, staging input is better or equal to texture in most cases. While WC-M and SM-M have similar results, opposite results are shown in II-M and KM-M. While the texture cache is designed to reduce the global memory bandwidth demand, its latency is still longer than shared memory. Therefore when there are long, complex computation phases with conditional branches and a large variance in input as in II-M, bandwidth is not a problem, and the short latency of shared memory makes SI much better. But in KM-M, featured with fixed-size input and almost equally long computation for every thread, the latency of texture fetch is well hidden and the GT mode wins because hardware cache brings the slightest overhead compared to explicit staging. However SI gradually catches up with GT when the thread block is larger, showing that the overhead of SI is finally hidden with more warps. Besides, MM-M’s GT mode shows superior performance over SI because in GT, row/column vectors can be cached with the hardware-managed replacement policy, while SI can only stage the row/column indices.

### 3.4.5 Reduce Results

Figure 3.4(f)-3.4(i) shows the Reduce kernel execution time for the two workloads with Reduce phase, WC and KM, with two different reduce strategies.
Overall, here staging input/output does not appear as helpful as for Map kernels, and we see less diversity in behavior across different workloads/strategies. In the majority of cases, G (or GT when texture is applicable) works the best. As can be seen from Table 3.2, the input-to-output ratio is quite large. Therefore, the benefit of staging output through shared memory cannot offset its overhead, and G (or GT) outperforms SO in all cases.

Staging input does appear to generate a considerable benefit in the case of KM-BR, by an average speedup of 2.25x over G. Here the size of each value type, i.e. a vector, is dependent on the dimension of the space and usually large enough to reach dozens of bytes. With G, multiple global memory transactions are needed for accessing each vector: data accessed for a half-warp at a time span across several 128-byte segments. Staging input improves performance by reducing the number of global memory transactions. This does not happen to WC, as the value type there, integer, makes global memory access already coalesced.

Comparing TR and BR, it appears that BR works better for KM, while TR performs better for WC. The reason lies in their input characteristics. KM has dozens of large distinct key sets, while WC has 10,000s of small ones. Therefore, TR achieves more parallelism with WC across key sets and BR with KM within each key set. This agrees with the finding from temporal data mining [13] that BR works better on small problem sizes, and TR on large problem sizes. The rather small number of key sets in KM, which limits the maximum parallelism of TR kernels, also explains the scalability problem observed in KM-TR. The G mode’s performance is flat, as an increased block size only makes more threads idle in the block. The SO mode’s performance degrades as more threads waste more cycles in synchronization, and bring no benefit due to the high input-to-output ratio.

Texture cache usage for two TR kernels gets an improvement of around 1.27x for WC-TR and around 1.12x for KM-TR over the G mode. Optimized for throughput requirement, the texture cache brings more benefit to WC-TR than to KM-TR. This is again caused by the aforementioned characteristics of the two TR kernels. WC-TR has a higher degree of parallelism, and hence more potential to take advantage of the texture cache.

3.4.6 Comparison with Mars

In this section, we compare our approach with Mars [28], the state-of-the-art, general-purpose MapReduce framework on GPU known by us.

Our framework and Mars share the same data transmission between host and device, as well as the same shuffle phase. The same Map and Reduce algorithms are implemented except for the Reduce phase of Word Count. WC in Mars does not have a Reduce phase since the number of occurrence of each word can be found as the number of records in each distinct key set after the shuffle phase. Yet for the purpose of making a fair comparison, we extended WC in Mars
Figure 3.5: Execution time composition of Mars and our framework in different memory usage modes at different problem sizes
by 1) enabling the Reduce phase, and 2) providing a Reduce function that follows a commonly accepted algorithm [23]. For the two workloads that have a Reduce phase, as Mars supports only thread-level reduction (TR), we only compare the TR implementation in our framework. For each memory usage mode, we use the optimal thread block size found in Figure 3.4.

First, we take a look at the impact of the five memory usage modes, enabled by our framework, on the Map and Reduce kernel performance. Figure 3.6 shows the speedup over Mars brought by all the memory usage modes in our framework. In most cases, even without using shared memory, our framework outperforms Mars by avoiding the two-pass execution. The G mode beats Mars by a factor of 2 at most (1.1 on average). The reason for the negative speedup in WC and SM is that both workloads have large quantities of results. In this situation, the two-pass running is better, which avoids the bottleneck in atomic updates in the global memory.

The G mode also delivers better performance for the two Reduce kernels, compared to Mars as well as to the SO mode. For the Map kernel, however, more advantage is obtained by our active use of shared memory as a staging area. In particular, with the help of output staging, the SO and SIO modes significantly outperform Mars, where G alone is not effective, as discussed above. The SIO modes, which has been shown in Section 3.4.4 to be overall effective, achieves an speedup between 1.3x and 3.73x over Mars, with an average of 2.67x. This demonstrates that SIO appears to be a good choice for Map kernels.

Then we examine the total MapReduce execution time of the compared systems (Mars, and the five memory usage modes with our framework), as shown in Figure 3.5. Each stacked bar gives the breakdown of separate phases in MapReduce: the Map, shuffle, and Reduce phases, as well as the I/O time spent in transferring data back and force between the GPU global memory and the CPU host memory. Here we show results from all three problem sizes that we tested: small, medium, and large. Note the reduce phases in SI and SIO modes are actually that of G
Figure 3.7: Kernel speedup by using GPU synchronization with yielding over non-yielding and SO respectively as input staging does not apply to the TR kernels.

The complete speedup is dampened by the portion of non-Map/Reduce phases in the end-to-end elapsed time. Word Count is an example where there is a large shuffle phase. Still, overall the G mode and SIO mode in our framework outperforms Mars by 34% and 64% on average, respectively. Yet a better approach is to adopt different memory modes in different phases adaptively, which we plan to explore in our future work. There are visible differences in the I/O time between Mars and our framework, due to that the data definition and indices can have slight difference.

3.4.7 Synchronization on GPU

Finally, we examine the effectiveness of our intra-block thread synchronization scheme, which allows waiting threads to yield using dummy global memory accesses. Figure 3.7 shows the benefit of such a scheme in busy waiting. Compared with the case where the waiting helper threads never yield, the kernel execution time improves between $-1.2\%$ and $13\%$ for all SIO Map kernels but MM-M: these kernels never go to global memory to fetch input data in the computation phase. Note that the benefit starts to appear after there are 128 threads within a block, when the GPU processor resource utilization is high enough, and the performance gain increases when the thread count per block grows. Yielding is especially beneficial for II-M,
which has long and complex computation phases.

3.5 Concluding remarks and Future Work

In this chapter, we discussed our approaches to overcoming the hurdles of enabling the aggressive utilization of GPU shared memory in MapReduce computation. We present a MapReduce framework that adaptively manages the limited shared memory space without requiring programmers to explicitly consider data placement in the GPU memory hierarchy. From our evaluation, we have found that despite its small size, the shared memory is worth exploiting and in particular, there is in most cases a significant performance gain when it is used to stage output data. Also, the Map phase of MapReduce workloads receives more benefit from using shared memory, mainly due to larger output sizes.

We plan to extend this work in the future. Most importantly, we will investigate solutions to make our framework autonomous, by enabling it to automatically and transparently adapt to individual workloads. Leveraging empirical observations and general guidelines found in this work, an intelligent MapReduce framework should be able to perform runtime, automatic configuration of parameters such as the shared memory space partition sizes and the thread block size. We also plan to extend our work to the newer GPU architecture, which has a global memory cache and offers the choice of user-configurable partitioning between the L1 cache and shared memory.
Chapter 4

Efficient Intranode Communication in GPU-Accelerated Systems

4.1 Introduction

In this chapter, we address the challenge of manually mixing MPI and GPU programming, by extending MPI and integrating direct support for the GPU memory space into MPICH2, a popular open-source MPI implementation. This interface allows programmers to pass GPU buffers directly to MPI routines without the need for explicit intermediate copies. As an initial step toward efficient MPI communication for GPU-accelerated parallel applications, we present an approach to perform efficient intranode communication. By integrating GPU data movement into MPI, multiple optimizations become possible. Temporary copies can be eliminated, freeing additional resources to the application and significantly improving performance. In addition, by pipelining GPU and inter-process data movement, PCIe and main memory concurrency can be leveraged to increase transfer efficiency.

We evaluate our system on several microbenchmarks and on a nine-point two-dimensional stencil benchmark that performs halo-type neighboring exchange communication. Results indicate up to twofold improvement in bandwidth for large messages and roughly 10% improvement in latency for small messages. These improvements in raw communication performance translated into an average improvement of 4.3% to the total execution time of the 2-D stencil benchmark across a range of process counts and problem sizes.

The rest of this section is organized as follows. Section 4.2 presents background information on GPU computing, MPI, and MPICH2’s intranode communication architecture. Section 4.3 discusses current challenges in mixed GPU+MPI programming. Section 4.4 introduces the design of our system, its integration with the Nemesis communication subsystem in MPICH2, and several performance optimizations. Section 4.5 presents an experimental evaluation. Section
4.6 summarizes our conclusions.

4.2 Background

This work focuses on enhancing the performance of intranode communication where the source, target, or both buffers reside in separate accelerator memory. In this section, we provide background on MPI.

4.2.1 The Message Passing Interface

MPI [10] is the industry standard for parallel programming. MPI defines point-to-point send/receive, one-sided, and collective communication operations and allows the programmer to construct parallel programs that are portable across virtually all parallel computing architectures. While MPI is best known for high-performance internode communication, most popular MPI implementations also provide highly optimized intranode communication between cores and processors on the same node.

4.2.2 MPICH2 Intranode Communication Architecture

MPICH2 is an open-source implementation of MPI standard. Designed for high-performance communication, MPICH2 is built on the Nemesis communication subsystem [17] and supports intranode message-passing through shared memory and internode communication by using network modules designed for specific networks. The Nemesis network module API efficiently supports high-performance RDMA-style networks as well as TCP. MPICH2 abstracts connections between two processes with a virtual channel data structure that stores state information associated with the connection. In this work, we focus on intranode GPU native communication and therefore will make modifications primarily to Nemesis.

MPICH2 has two data transmission modes: eager mode and rendezvous mode. Eager mode is intended for shorter messages and is optimized for latency; rendezvous mode is intended for large messages and is optimized for bandwidth. In eager mode, a message is sent to the other party through shared-memory message queues. The data is copied from the user buffer into one or more available elements in a free queue, which are then inserted to the receiver’s receive queue. The message queue is implemented with atomic memory operations so that multiple processes can enqueue elements on the queue concurrently without locking. This strategy means that only one receive queue is needed per process, rather than one queue per pair of processes.

Nemesis uses the large message transfer (LMT) protocol to implement rendezvous mode. In the LMT protocol, shared-memory copy buffers are created between pairs of processes the first time they communicate with each other. The copy buffers are arranged as a ring buffer.
double *dev_buf, *host_buf;
cudaMalloc(&dev_buf, size);
cudaMallocHost(&host_buf, size);

if (my_rank == sender) {
    /* sender */
    computation_on_GPU(dev_buf);
    cudaMemcpy(host_buf, dev_buf, size, ...);
    MPI_Send(host_buf, size, ...);
} else {
    /* receiver */
    MPI_Recv(host_buf, size, ...);
    cudaMemcpy(dev_buf, host_buf, size, ...);
    computation_on_GPU(dev_buf);
}

Listing 4.1: Example MPI program with manual data movement.

so that the sender can copy into one buffer while the receiver is copying out of another buffer in a pipelined manner. Nemesis supports other implementations of LMT that use vmsplice or I/OAT with kernel module support; however, these methods are not considered in this work. The Nemesis progress engine is responsible for driving the LMT protocol.

4.3 Challenges in CUDA+MPI Programming

The mixed CUDA+MPI parallel programming model has gained widespread adoption for programming clusters with GPU accelerators today. In this model, intranode parallelism is expressed in the CUDA model, and internode parallelism is managed through MPI. Because MPI implementations are not aware of the distinct accelerator memory space, the programmer must manually copy data between device and host buffers, as shown in Listing 4.1. In this simple example, the only purpose of the host_buf buffer is to facilitate MPI communication of data stored in device memory. As the number of accelerators (and hence distinct memories) per node increases, manual data movement poses significant productivity problems [70].

Beyond the problem of programming productivity, the performance of such a mixed programming scheme is also troublesome. In Figure 4.1, we measured the latency and the bandwidth of intranode communication between two MPI processes running on a single machine, using the latency and bandwidth tests from the OSU MPI micro-benchmark suite [12]. In this experiment, we vary the location of source and destination buffers between main memory and GPU device memory. When a buffer is located in device memory, a manual cudaMemcpy is performed between a temporary host buffer and the source/target device buffer.

From these results, we see that latency is increased and bandwidth decreased by more than an order of magnitude for small messages when either buffer is located in device memory. The
bandwidth achieved when a GPU buffer is used peaks at between 1.6 and 2.8 GB/sec. This is significantly lower than the theoretical peak of this system, 6 GB/sec, which is bounded by the smaller of the PCIe and memory bandwidths. This gap of roughly 50% in performance is incurred because the naïve two-copy method ignores parallelism present in the hardware that allows bidirectional memory copies between host and device and host-to-host copies to proceed in parallel. As we describe in Section 4.4, techniques such as pipelining of data transfers can greatly improve communication performance by leveraging hardware data movement parallelism. Such techniques can be implemented in an application by the programmer; however, they introduce significant complexity and, as we will demonstrate, can deliver greater performance benefit when integrated with the existing intranode communication infrastructure.

4.4 Design

To address the challenges of mixed CUDA+MPI programming, we integrate support for device-resident buffers into the MPI implementation. This approach not only improves productivity but also greatly improves performance by more efficiently communicating data within the node. The GPU memory buffer support is added to MPICH2 by making MPI communication routines aware of addresses in the device memory, enabling MPI to internally perform efficient intranode data movement. DMA-assisted asynchronous PCIe data movement is leveraged and built into the current LMT pipeline of Nemesis in order to increase throughput. Data movement between host and device also plays an important role in achieving a high performance implementation,
and several schemes are explored.

### 4.4.1 Integrating GPU-Awareness in the MPI Interface

Several options are possible to integrate support passing GPU buffer arguments to MPI routines. CUDA provides a unified virtual address space (UVA), which maps device buffers into the virtual address space and provides routines that can be used to query whether a given address corresponds to host or device memory. This way, device buffers can be passed directly to MPI, and MPI can decide whether to start a PCIe or a main memory transfer by internally querying the buffer’s location. However, other accelerator programming models of interest, such as OpenCL [38], do not provide a unified virtual address space and require several handles (e.g., context handle) to access the device.

To provide a fully generic interface that can support a variety of accelerator models, we have extended the MPI interface with an \texttt{MPIGPU} interface that adds a buffer type parameter for each buffer passed to an MPI routine. The buffer type parameter allows the programmer to specify how the corresponding \texttt{void *} argument should be interpreted and in which memory space the argument resides. For example, in an OpenCL program, a struct or record argument with relevant resource handles would be passed, which provides MPI with all the information needed to access the given buffer.

This new interface is demonstrated in the following example, where a nonblocking receive is posted to receive into a host buffer and a send is performed from a GPU buffer.

\begin{verbatim}
MPIGPU_Irecv(host_buf, MPIGPU_HOST_CPU, count,
            datatype, left_neighbor, tag, comm, &request);
MPIGPU_Send(dev_buf, MPIGPU_BUF_GPU, count,
            datatype, right_neighbor, tag, comm);
\end{verbatim}

### 4.4.2 Eliminating Unnecessary Copies

Integrating support for GPU buffers into the MPI interface not only enhances productivity but also can significantly improve performance by allowing the implementation to eliminate unnecessary copying of data into temporary buffers. In Figure 4.2(a) we show the data movement corresponding to the manual mixed CUDA+MPI code from Listing 4.1. In this example, which is currently how all CUDA+MPI codes are written, a user-managed host-side buffer is used to help transfer data buffers residing in GPU memory. Data is first transferred from the device to a temporary buffer; next Nemesis copies this data into an internal buffer that is accessible to both processes; it is copied again into a temporary user buffer; and finally it is transferred to the device. Thus, when both the source and destination buffers reside on the GPU, two memory
copies and two PCIe data transfers are involved.

By integrating support for GPU buffers into MPI, we can eliminate the unnecessary user-level staging of data between MPI and the device, as shown in Figure 4.2(b). Now MPI takes the responsibility for moving data from device memory. It will first check the buffer type, and then transport the given data directly to the shared buffer used for Nemesis interprocess communication. As we demonstrate in Section 4.5.1, this optimization significantly reduces the latency of communication operations, especially for latency-sensitive small messages.

4.4.3 Increasing Throughput with Pipelining

While eliminating memory copies is extremely beneficial for small messages transported by using the eager mode protocol, optimizing the rendezvous mode protocol is the key to providing good bandwidth for large messages. The LMT protocol for rendezvous mode in Nemesis is implemented through a shared-memory copy buffer, which is created in a virtual channel between every pair of MPI processes in a given node. The copy buffer is managed as a ring buffer, as shown in Figure 4.3(a), where a copy buffer length for each buffer unit represents its full/empty state.
When a message is about to be transported through this ring buffer, the sender packs the message data into multiple segments and tries to copy them into as many consecutive available ring buffer units as possible; the receiver serially unpacks the consecutive buffer units out of the ring buffer and into the user-designated receive buffer. The data segmentation in Nemesis is done according to the maximum length of a copy buffer unit and the MPI data type. In MPICH2, a progress engine drives this in a nonblocking manner—the packing/unpacking progress can be paused, if the next buffer unit is unavailable, and resumed later; the progress engine can take the chance to drive other pending actions.

Modern GPUs are equipped with one or more DMA engines, which can be used to either synchronously or asynchronously transmit data between GPU device and page-locked main memory. Asynchronous DMA transfer has been shown to maximize the performance of the PCIe bus and achieve the highest bandwidth [50]. Thus, the Nemesis LMT shared buffers are page-locked to enable asynchronous DMA copies.

However, a challenge brought by asynchronous GPU memory copy in a nonblocking progress-engine-driven loop is, for both sender and receiver, to poll the ring buffer and decide whether to pause the loop. Originally, a CPU-side memory copy is a synchronous action and returns with data ready in the destination. But now, we need to poll the status of both (1) availability of the buffer units, which are changed by the other party in a communication pair, and (2) completion of previously submitted asynchronous DMA memory copy. The two types of polling activity are interdependent: failing to poll the completion of DMA copy will delay the notification of availability of buffer units to the other party, and failing to check the newly available buffer units will delay starting new asynchronous DMA copy.
Under these requirements, we modify the algorithm (in the loop of sender and receiver both) by including two new *ahead* pointer (*Ra* and *Sa* in Figure 4.3(a)) and start DMA asynchronous copies as eagerly as possible. That is, whenever there is an available buffer (empty for sender, or nonempty for receiver), a new DMA copy is issued, and the *ahead* pointer (*Sa* for sender, *Ra* for receiver) proceeds; but the length of that buffer unit is not changed until the completion of the DMA command. Here we put priority on availability of buffer units over timely checking DMA completion. This strategy keeps the buffering throughput efficient, since the PCIe latency is much larger than that of main memory operations, and floods the DMA engine, which typically can combine DMA requests or at least reduce idle time by pipelining operations. The pseudo-code for the receive progress loop is shown in Algorithm 1; the send-side loop is similar. Note this is a multilevel nested loop, and the loading/saving-state activities (line 1 and 5) are designed for the re-entrant progress driven by the progress engine. The unpack function (line 14) eagerly starts the DMA asynchronous memory copy and handles data layout according to MPI types. The inner loop (line 3–11) holds the progression of *Ra* under either of the following circumstances: (1) there is more data to receive, but the sender has not put data in the buffer, or the pending DMA commands has reached a maximum threshold; or (2) there is no more data to receive, and there are pending DMA commands. The DMA maximum pending threshold is a parameter to decide the level of compromising, for the eagerness of initiating DMA copies, the timeliness of checking DMA completion, and, in turn, the timeliness of notifying the availability of buffer units to the other party.

With a specific MPI data type, a single element may cross two neighboring units. However, data segment pack/unpack functions work only at the boundaries of fundamental data types. Thus, surfeit data, which belongs to the next data element, may not be copied out of the shared memory buffer. The surfeit is decided upon return of **unpack**, even if its asynchronous DMA request is proceeding. When detected, it can be immediately moved to the end of this unit, making it a contiguous buffer with the available data in the next buffer unit. Thus, *Ra* can proceed proactively. Later, when a DMA command is found complete, the previous buffer unit’s length can be set to 0. This is shown in Figure 4.3(b). Also note that this surfeit occurring in the buffer unit at *R*−1 can make us lose one unit, because the postponed DMA polling for *R*, due to proactive DMA initiation, may not clear its buffer length in time, if the DMA maximum pending threshold is set to the number of buffer units.

### 4.4.4 Efficient Host-Device Data Movement

GPU memory, the on-board device memory, is separated from the main memory on the host. In spite of the recent CUDA UVA and GPUDirect technologies [50], special GPU memory copy commands, for example, **cudaMemcpy** (which are different from **memcpy** on the host side),
Algorithm 1: Pseudo-code of receive progress loop

1 load state;
2 repeat
3   while (more data AND (copybuf_len[Ra] == 0 OR pending dma reaches max)) OR
     (no more data AND pending dma) do
4       if failed for a certain number of polls then
5         save state and exit;
6       while query dma of R finish do
7         if surfeit[R-1] ! = 0 then
8           copybuf_len[R-1] := 0;
9         if surfeit[R] = 0 then
10           copybuf_len[R] := 0;
11         proceed R;
12       if more data then
13         proceed Ra;
14         unpack(buf[Ra] - surfeit, &surfeit);
15         surfeit[Ra] := surfeit;
16         if surfeit ! = 0 then
17         copy surfeit buf before buf[Ra+1];
18     until no more data AND no pending DMA;
must be used to handle the memory copies involving GPU memory. These commands utilize
the DMA engine on the GPU through interaction with the GPU driver. Typically these GPU
memory copy commands ask for the location of source and destination buffers, in main memory
or GPU device memory, as function arguments. The different memory copying methods make it
challenging to introduce GPU memory awareness to MPICH2, which assumes a single memory
space. When a memory copy needs to be initiated, the locations of source and destination
buffers must be known beforehand to determine whether normal memcpy or GPU memory copy
should be used.

The recent CUDA UVA technology provides programmers with a logical universal virtual
address space to make programming easier. It maps GPU device memory buffers to a part of
the virtual address space of a process so that the driver and the runtime library can tell where
a buffer is located using its address. This provides developers with a pointer query API and
an easier version of cudaMemcpy, the latter of which, asking for only a default argument, can
determine the location of a buffer by itself and perform a transparent memory copy. Though
the proprietary driver is a black box, we speculate that the default version of cudaMemcpy API
embeds something similar to that done for the pointer query API before starting the actual
cudaMemcpy action with correct buffer location parameters.

Therefore, adding support for GPU memory copy into MPICH2 can be done in three ways:

- **UVA-Default**: This method simply replaces all memcpy with cudaMemcpy with the default
  parameter, relying on UVA and the GPU driver to perform device-host and host-host copy
  operations. This is the easiest method for extending MPICH2.

- **Query-and-memcpy**: This method queries the buffer location with the driver at run
time. After a query, it can be decided whether to use normal memcpy or a GPU memory
  copy command.

- **Parameterized-memcpy**: This method stores the location of a buffer pointer and passes
  it around with the pointer within MPICH2. Any memory copy that may involve a GPU-
  resident pointer will be modified to check this location first and decide to use normal
  memcpy or a GPU memory copy command.

The first two approaches require less modification to MPICH2’s internals. However, they
may introduce extra overhead of interaction with the GPU driver. In Figure 4.4, we measured
the bandwidth of moving data across host and a GPU device with all three methods (using the
same system setup as in Section 4.5). While the overhead is less important when the copied data
sizes are increased, for data sizes smaller than 64 KB, Query-and-memcpy shows this overhead,
more clearly in the direction of “Host to Device.” Interestingly, the UVA-Default method does
not show any overhead. The difference here is that the Query-and-memcpy method enters the
Figure 4.4: Bandwidth of different GPU memory copy methods between host and GPU device. Adapted from CUDA SDK bandwidth test [11]. Host buffer is pinned always.

driver twice, for pointer querying and the GPU memory copy command, whereas UVA-Default and Parameterized-memcpy do so only once and probably cause less operating system overhead due to privilege level change.

Although UVA-Default seems to be a good option, a factor that one cannot neglect is the performance on the host-side memory copy, if it were used to replace all memcpy in MPICH2. The CPU-side memcpy in MPICH2, namely, MPIU_Memcpy, is actually an efficient implementation [16] that tries to optimize both memory copy performance and the impact on cache behavior of the computation. When an MPI communication primitive is called, it may evict the data stored in CPU cache, which in turn affects the performance of computation after the copy completes. Therefore, the less such cache disruption an MPI implementation does, the more easily an MPI program developer can analyze the performance of his own code. In MPICH2, MPIU_Memcpy uses nontemporal instructions to alleviate the impact for large enough copy data sizes (i.e., 64 KB and beyond).

In Figure 4.5, we show the bandwidth of host-side memory copy using UVA-Default, MPIU_Memcpy, and memcpy (from glibc), on the same system as above. As MPIU_Memcpy falls back to memcpy before 64KB, they overlap until non-temporal version is used. The bandwidth of memcpy in the mid-range is larger than the actual memory bandwidth, which is measured in larger sizes, and is due to the hardware cache behavior of the microbenchmark. The true memory bandwidth is shown when MPIU_Memcpy starts the nontemporal implementation but appears more gradually in memcpy. However, UVA-Default does not behave as well as the other two. The reason is probably that it does too much work in checking the location of destination and source buffers, and this work disturbs hardware cache a lot even for small data copy sizes. This
overhead is not shown in the previous experiment, however, because the PCIe latency dominates it in that case but here memory and cache latency is smaller.

In addition to performance considerations, we decided to adopt Parameterized-memcpy for portability reasons to support GPU programming models that do not provide UVA.

4.5 Evaluation

Our evaluation in this section and the previous sections is conducted on Keeneland [69] cluster, a National Science Foundation Track2D Experimental System based on the HP SL390 powered with Nvidia Tesla M2070 GPUs in Oak Ridge National Laboratory. Each compute node in Keeneland has two Intel Xeon X5660 CPUs, 24 GB main memory, 3 GPU devices connected through 2 IO hubs; nodes are connected via single rail, QDR Infiniband. The software environment is CentOS release 5.5 (Final) with Linux kernel 2.6.18-194.el5.perfctr, CUDA driver/runtime v4.0.

4.5.1 Exploration of Buffering and Message Queue Parameters

We present an exploration of the parameter space for the shared buffer and message queue. Using the ideal parameters for our experimental platform, we measure performance and compare it with a baseline manual code similar to that presented in Listing 4.1. Performance measurements were taken using the latency and bandwidth tests from OSU microbenchmark suite [12].

First, we determine the shared buffer unit size for LMT shared buffer using bandwidth test.
The results are shown in Figure 4.6 for 4 MB, 1 MB, and 256 KB messages, which are points of interest for MPI messaging and are representative of trend. First, we notice that different trends are shown when GPU memory is used or not (as we compare “H2H” to “D2D”/“H2D”); this result suggests that different parameters should be used for host-only and host-GPU transfers. Second, as long as GPU memory is used, on one side or both, the best bandwidth can be attained only with a 128 KB or larger shared buffer unit size, since small-sized data transmission on PCIe bus is inefficient. The optimal point occurs at 256 KB, with the exception of 128 KB for 256 KB messages. Considering the possibility of even larger messages, we chose 256 KB for this parameter. Third, for host-only messages, the peak bandwidth is reached with smaller unit sizes (32 KB by default now in Nemesis). The reason is that because main memory has much lower access latency than does the PCIe bus, moving fine-grained chunks in shared buffer can increase parallelism of the pipeline. Therefore, we should let the shared buffer fall back to its original parameter value when no GPU memory is involved. And this information regarding the location of buffer of the other side can be exchanged in the LMT’s handshaking step.

Using this shared buffer parameter setting, we try to cap the performance curves of eager message queues, in order to find the best message queue element sizes and the threshold of switching modes. In Figure 4.7, latency and bandwidth of message queues at 16 KB, 32 KB, 64 KB, and 128 KB are shown together with that of the shared buffer. The latency of the message queue has no significant advantage over that of the shared buffer when the GPU buffer is used, because the PCIe bus latency is dominant. However, the message queue can have a larger number
of cells than the number of shared buffer unit sizes, because of its scalable linear-complexity nature. Therefore it shows better bandwidth, simply because it can hold more simultaneous requests. With regard to latency or bandwidth, the message queue loses to the shared buffer beyond the point of 64 KB, which we choose as the threshold of switching modes. When the message queue is used for sizes below that, we choose 64 KB as the message queue element size, which is also the original value now in the system for host-side messages.

4.5.2 Performance Comparison with Manual CUDA+MPI

Using this set of parameters, we compare the latency and bandwidth of MPIGPU with manual CUDA+MPI where the user must manually move data between host and device in Figure 4.8. The benefit of latency for small messages, from eliminating one or two main-memory copy, is bounded by the memory latency. On average, for messages smaller than or equal to 64 KB, latency improvement is 6.4%, 15.7%, and 10.9% for “D2D,” “H2D,” and “D2H,” respectively. With full range of size till 4 MB, the improvement is increased to 24.5%, 31.9%, and 23.9%, respectively. The benefit of bandwidth increases with messages sizes and achieves up to 2x speedup over manual mixing CUDA with MPI for all three cases shown. On average, for messages between 64 KB and 4 MB, the improvement is 56.5%, 48.7%, and 27.9% for “D2D,” “H2D,” and “D2H,” respectively; with full range of size between 1 byte and 4 MB, the improvement is 18.1%, 27.5%, and 10.2%, respectively. While the peak “D2D” bandwidth is around 60% of the theoretical one—that is, the smaller of bandwidth on PCIe and main memory assuming a fully pipelined conduit—the “D2H” bandwidth efficiency is nearly 90%, almost saturating the
## 4.5.3 Evaluation with 2-D Stencil Benchmark

Stencil2D from SHOC benchmark suite [20] measures the performance of a halo-type, nine-point, two-dimensional stencil computation. It performs an iterative stencil computation on the GPU and requires a halo exchange every \(\text{haloWidth}\) iterations. In this type of computation, processes are arranged in an \(N\)-dimensional Cartesian grid, and each process is assigned a corresponding section of a \(N\)-d array. Periodically, a process must obtain the values that its neighbors have calculated for the array elements that neighbor its patch, or its halo. Thus, this communication idiom, which is common across a broad range of iterative solvers, is referred to as a halo exchange.

In Figure 4.9, the mean time speedup of using MPIGPU against the original manual version is shown. Also shown is the percentage of execution time spent on communication. In general, we have seen an average 4.3% improvement in the total execution time of stencil2d. Grouped by different number of processes, an averaged improvement of 5.8%, 5.6%, and 5.0% are shown for 4, 6, and 8 processes respectively, except for the case of two processes, where only a 0.69% improvement is shown. This is probably because Stencil2D exchanges a vector-typed data along one dimension, which leads to a number of small PCIe requests. Thus, more processes leads to more asynchronous data transmission commands along this dimension and hence a larger po-
potential for DMA engines to combine these small requests than in the case of only two processes. Also note that the percentage of the execution time spent in communication decreases when we increase the problem size, because in Stencil2D the computation grows quadratically with the problem size and the communication grows linearly. This limits the MPIGPU’s potential benefit when we increase the problem size. Since a node has only three GPU devices, when the number of processes is larger than that of the devices, the tasks are assigned in a round-robin order. And a context-sharing overhead is introduced when multiple processes are using the same GPU device. This is the reason the 8-process cases have a smaller percentage of communication than do the others (although they have more data to exchange) and hence less improvement, shown in the medium and large problem size. The trend is not shown in the small problem size, where the tiny (6144 bytes) exchanges data size and the smaller number of noncontiguous PCIe data transfers cause other latency-induced overhead.

4.6 Concluding Remarks

GPU and other accelerators are emerging computation hardware in HPC systems. In the past few years we have seen rapid evolution of GPUs, with new architectural functionality and software features every season or half year. With increasing interest in these accelerators, we expect systems softwares, such as the MPI communication system, to have a more pressing need to extend native support of GPU and other accelerators for application development.

In this work, we propose a uniform MPI communication interface for a GPU-accelerated system, where user-specified buffers can reside in traditional main memory or in GPU mem-
ory. We present the design and implementation of integrating transparent GPU-awareness into MPICH2, a popular MPI implementation. With a focus on intranode communication, we further improve the throughput by taking advantage of the DMA engines on GPU and adapting the ring buffer pipeline of Nemesis communication subsystem. We also highlight the importance of choosing an efficient memory copy method. The augmented MPICH2-GPU system is evaluated through both microbenchmarking and a halo exchange benchmark and shows 23.9–31.9% latency improvement and 10.2–27.5% bandwidth improvement, for different source and destination buffer locations, and 4.3% average performance improvement in the halo exchange benchmark.
Chapter 5

DMA-Assisted, Intranode Communication in GPU Accelerated Systems

5.1 Introduction

The processing units on GPUs can only operate on data that is located on the GPU on-board memory. While the GPU programming libraries provide mechanisms for transferring data between host memory and GPU memory, the challenge remains to efficiently schedule and synchronize these transfers. Managing data transfers has been previously studied [63, 70, 71] with respect to message passing libraries, specifically those implementing the Message Passing Interface (MPI) standard [10]. These studies concentrated on optimizing transfers between GPU memory and host memory, even when the source and/or destination of the data was memory on a GPU device. One of the shortcomings of these previous approaches is that they did not take advantage of systems where multiple GPUs were installed in the same compute node and data had to be moved between them. In Chapter 4, we proposed a shared-memory approach that utilizes a common host memory buffer that is visible to both the source and destination process to reduce the number of memory copy operations required, thus improving overall performance. However, this approach still requires intervention from the host processor and memory to “stage” data before it can be moved between the two GPU devices. In this chapter, we introduce the mechanism of using GPU DMA engines to expedite intranode MPI data movement.

Recently, a new GPU IPC feature has been introduced on new GPU hardware from NVIDIA, that allows GPU direct memory access (DMA) engines to directly move data from one GPU to another on the same node. In this chapter, we explore the use of the GPU’s on-board DMA
engines, cross-GPU peer-to-peer direct accessibility, and GPU IPC on modern GPUs to design an efficient intranode cross-GPU/CPU peer-to-peer communication scheme for MPI communication. Taking advantage of these latest features, our scheme can bypass the unnecessary data paths through host memory used in current intranode GPU communication mechanisms for MPI, and can instead realize the direct transfer between source and destinations buffers in the best case. The contribution of this work includes the following aspects.

- We introduce the usage of GPU DMA engines, GPUDirect [7] and cudaIPC [6] in designing a DMA-assisted peer-to-peer direct intranode MPI communication subsystem. We further show that this design can be extended to communications between a CPU and a GPU device.

- We study the different design choices in our system, including protocol designs and DMA engine usages.

- We implement our design by adapting MPICH, a widely-used MPI implementation. We evaluate the performance on two typical GPU-accelerated systems. Our results show that the DMA-assisted peer-to-peer communication is beneficial, especially when communication participant GPU devices are close in a system. Applying the technique to Stencil2D, a stencil computing benchmark from SHOC [20], it shows on average 4.7% and 2.3% performance for single-precision and double-precision runs respectively.

The rest of this section is organized as follows. Section 5.2 presents background information on GPU computing, MPI, and MPICH’s intranode communication architecture. Section 5.3 introduces the design of our system, its integration in MPICH, protocol designs, and a memory handle caching optimization. Section 5.4 presents an experimental evaluation. We conclude our findings in Section 5.5.

5.2 Background

5.2.1 MPI and MPICH

MPI [10] is the industry standard for parallel programming on virtually all parallel computing architectures. Most popular MPI implementations provide highly optimized internode communication as well as intranode communication between cores and processors on the same node. MPICH, developed at Argonne National Laboratory, is a widely used open-source MPI implementation. Its intranode communication is handled by the Nemesis [17] communication subsystem. MPICH has two data transmission modes: eager mode, optimized toward latency for shorter messages, and rendezvous mode, optimized toward bandwidth for large messages.
The rendezvous mode is implemented through the *large message transfer* (LMT) protocol in Nemesis. Currently, this protocol has several different implementations including using shared-memory buffers and kernel-assisted single copy using DMA. The shared-memory buffer implementation allocates buffers shared between the sender and receiver processes for them to store/remove message data. The sender and receiver processes work in parallel to pipeline the memory copies.

In our previous work, we designed an approach to allow intranode communication from GPU buffers [34]. This eliminated the need for the application to explicitly copy data from the GPU memory to the host memory before calling the MPI communication operation. The shared-memory LMT implementation was modified to use GPU data movement commands to directly copy data into an LMT buffer. However, this method still requires copying GPU-resident data to the shared buffer in host memory. This approach requires two DMA transfers and intervention from the host processor and memory for the data transfer to occur.

### 5.3 Design

In this section, we describe our design for DMA-assisted intranode MPI point-to-point communication between GPU memory buffers.

#### 5.3.1 DMA-assisted Intranode Communication

Each communicating process within a node has its own virtual address space in CPU and GPU memories. A virtual address of one process cannot be dereferenced in the address space of another, without OS support for sharing memory mappings. While peer-to-peer GPU memory copies (via *GPUDirect*) are possible with CUDA, they are restricted to a single process, and does not work between processes. In previous work, we solved this problem by performing MPI communication between GPUs via host-side shared memory (*shm*) by extending MPICH’s Nemesis communication system.

As mentioned earlier in Section 2.4, recently CUDA (v4.1 or later) has exposed a new family of IPC functions, namely *cudaIpc*, that can be used to export a memory handle to a GPU memory allocation from one process directly into the address space of another process within the same node [6]. The MPI process driving the communication can then issue an asynchronous DMA request, by calling *cudaMemcpyAsync*, to move data between the participating GPUs directly. This feature, together with *GPUDirect*, can be used subsequently to perform peer-to-peer data transfer. This allows us to completely avoid pipelining through host-side shared memory buffers. It is important to note, however, that *GPUDirect* is limited only to peer GPU devices that are connected to the same I/O hub; such “peer accessibility” can be queried from
the GPU device. In our design, we use this approach for peer GPU devices, and fall back to the original shared memory based approach for other GPU devices.

As no static binding exists between an MPI rank and a GPU device, the process can choose any available GPU at runtime. Therefore a process cannot know whether peer accessibility is available to the pair by using its own information. To solve this problem, we use the handshake phase of Nemesis’ LMT protocol (discussed in Section 5.3.2) to exchange the peer accessibility information of the devices before performing the communication. Note that the usage of LMT limits the applicability of DMA-assisted communication only to MPICH rendezvous mode, which is primarily used for large messages.

5.3.2 LMT Protocol for Peer-GPU Communication

In Nemesis, three LMT protocol models, PUT, GET, COOPERATE, are present as a guidance for intranode communication implementers. PUT and GET protocols are used to implement kernel-assisted single-copy approach, and the COOPERATE protocol is used for the shared memory based intranode communication. The three protocol models are different in the process that initiates the payload transfer. In this work, we show our design of LMT-Peer-GPU protocol, which can adaptively change into a PUT, GET or COOPERATIVE mode, depending on the peer accessibility. Figure 5.1 shows the overall control flow.

When the sender starts to participate in the handshake, it retrieves the inter-process memory handle for the sender’s data buffer and then sends this memory handle along with the device number, packaged in a cookie, with the Request-To-Send (RTS) message to the receiver. When the RTS message arrives at the receiver process, it inspects the cookie and checks the peer accessibility of the two devices. If GPU peer accessibility is available, it performs peer-to-peer GPU communication by using one of the three LMT protocols. Otherwise, it reverts to the shm approach. In another cookie created for the Clear-To-Send (CTS) message, the receiver embeds this decision, in order to inform the sender about it.

If the source and destination GPUs are peer accessible, the receiver can choose one of the three modes, namely PUT, GET, COOPERATE. This choice is arbitrary when GPU peers are accessible in both directions, but not in special cases (as further explained in Section 5.3.3).

**LMT-Peer-GPU-Get** If the receiver decides to GET the data, after receiving the RTS, it opens the sender’s memory handle, maps it into its address space and starts peer-GPU data movement. A progress element is inserted into MPICH’s progress engine queue by the receiver and the DMA status is polled for completion. A DONE message is then sent to notify the sender of completion.
LMT-Peer-GPU-PUT  If the receiver decides to let the sender push the data, it retrieves the interprocess memory handle of the receive buffer, packages it with the device number as well as the decision in the CTS packet’s cookie, and sends it back to the sender. The sender opens the receiver’s memory handle, maps it into its address space and executes peer-GPU data transfer. A progress elements is inserted by the sender. The progress engine is again polled for completion, and a DONE message is sent to notify the receiver of completion.

LMT-Peer-GPU-Coop  If the receiver decides both sides can help the data transfer, after two interprocess memory handles of the other process are exchanged similarly in RTS and CTS, the payload is divided into two halves; the sender puts one half in the receive buffer and the receiver gets the other half from the sender’s buffer. Progress elements are created on both sides. When receiver is done, a COOKIE message is sent to the sender to notify the partial completion of the data transfer; the sender finally sends a DONE message to denote full completion.

Arbitrary as the receiver’s decision may be in a mutually accessible case, choice of the protocol decides the GPU, and hence the DMA engine, that will be used for data movement, i.e., the driving process will use the DMA engine on its GPU. Also, DMA requests issued to the same engine will be serialized, and thus the choice of the protocol can be very critical in removing or creating DMA contention, depending on a certain communication pattern.
5.3.3 Intranode MPI Communication between Host and Device

Aside from GPU-to-GPU communication, using GPU IPC, we can perform intranode MPI communication between the host memory and the device memory. As a current limitation, only the GPU device memory can be exported to another process in `cudaIpc`. Main memory buffers cannot be exported without the support of kernel modules. This means that the process with its communication buffer in the main memory (the *host-side process*) must be the one to initiate the payload transfer. Upon receiving the interprocess memory handle exported from the *device-side process*, the host-side process opens it and maps the memory to its address space and then initiates the data transfer (to or from the device).

The host-side process needs a valid GPU context to request the GPU DMA engine for data communication between the host and the device. However, this may not always be the case. Depending on the availability of an active GPU context, two situations might arise for host-device MPI communication, as described below.

**Attach** If no active GPU context is available, the host-side process can *attach* to any available GPU device, by creating a new context on that GPU. A good choice here is to attach to the same device that contains the communicating device buffer. The context is then cached and can be reused for future data transfers.

**Relay** If an active GPU context is already available, the DMA engine of the corresponding GPU can act as a *relay* to perform the data transfers with the device-side process. Though we can temporarily change the active GPU context to use another GPU device, possibly the one with the communicating device buffer, and change it back after the communication is done, we tend not to do this. Since the active device context is a global setting in CUDA, changing it will redirect all GPU commands issued simultaneously to this communication onto the temporary active device, potentially polluting the user’s program.

5.3.4 Efficient Memory Handle Management

In all the LMT peer-GPU communication protocols, we first get the inter-process memory handle of a memory region in one process, then open it and map it to the address space of the other process. While getting the memory handle (`cudaIpcGetMemHandle`) is a light-weight operation [50], we find that opening the handle (`cudaIpcOpenMemHandle`) is expensive, probably due to interactions involving the importing and exporting devices and the driver run on the host side.

In a preliminary design, we open a memory handle after the RTS/CTS message exchange at the beginning of the communication, and close it after the data transfer is done. Repeated
opening and closing of the interprocess memory handle causes significant performance overhead.

Observing that many GPU programs have a relatively fixed memory creation pattern, e.g. creating a large memory region before computation, reusing it for computation and data communication, and only releasing it after a period long enough, we choose to cache memory handles. Therefore, when an communication is done, we do not close that memory handle and leave it open. Next time, when a memory handle arrives in an RTS/CTS packet’s cookie, we check first if the memory handle has been opened locally. If it is the case, our memory handle caching eliminates the reopening/closing memory handles. We observe the latency is more than halved after applying this technique.

Yet this design leads to another problem in closing a memory handle. Since we always leave it open when an LMT protocol is finished, we cannot know when a memory handle should be closed. And closing a memory handle should happen before the memory region is actually freed [50]. To solve this problem, we add a two-phase GPU memory free mechanism, by providing two function APIs (gpuMemFree and gpuMemFree_commit). When a GPU memory free is called upon a memory region, it is only recorded, with its memory handles marked if it is exported ever, but not released immediately. When a GPU memory free commit is called, the marked memory handles will be exchanged with other processes on that node. If found, a process will close the memory handle. After all finish closing memory handles, GPU memory regions are actually released.

5.4 Evaluation

5.4.1 Experimental Setup

Our evaluation is conducted on two typical GPU-accelerated clusters. Keeneland [69] cluster, a National Science Foundation Track2D experimental system based on the HP SL390 in Oak Ridge National Laboratory, is powered with NVIDIA Tesla M2070 GPUs. Each compute node in Keeneland has two Intel Xeon X5660 hex-core CPUs, 24 GB main memory, 3 GPU devices connected through 2 IO hubs; nodes are connected via single rail, QDR InfiniBand. The software environment is CentOS release 5.5 (Final) with Linux kernel 2.6.18-194.el5.perfctr, CUDA driver/runtime v4.1. Magellan [4] cluster, a DOE mid-range distributed computing research effort in Argonne National Laboratory, is powered with NVIDIA Tesla M2070 GPUs. The GPU nodes in Magellan has each four AMD Opteron(tm) Processor 6128 quad-core CPUs, 64 GB main memory, and 2 GPU devices connected to 2 IO hubs; nodes are connected via QDR InfiniBand. The software environment is CentOS Linux release 6.0 (Final) with Linux kernel 2.6.32-71.29.1.el6.x86_64, CUDA driver/runtime v4.1.

These two systems represent the typical multi-GPU heterogeneous architecture today, with
different cross-socket interconnect, NUMA settings and GPU connection topologies, as summarized in Table 5.1. Performance measurements were taken using the latency and bandwidth tests from OSU benchmark suite [12] in the following sections, unless stated otherwise.

Table 5.1: Keeneland and Magellan Architecture and GPU Topology

<table>
<thead>
<tr>
<th>Cluster</th>
<th>NUMA nodes</th>
<th>Interconnect</th>
<th>GPUs</th>
<th>GPU Topology</th>
<th>Peer Access</th>
<th>Distance between Peers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keeneland</td>
<td>2</td>
<td>Intel QPI</td>
<td>3</td>
<td>GPU 0: Node 0; GPU 1,2: Node 1</td>
<td>Only GPU 1 and 2</td>
<td>a sharing IO Hub</td>
</tr>
<tr>
<td>Magellan</td>
<td>4</td>
<td>AMD HT</td>
<td>2</td>
<td>GPU 0: Node 0; GPU 1: Node 3</td>
<td>Yes</td>
<td>2 HT hops</td>
</tr>
</tbody>
</table>

5.4.2 DMA-assisted Intranode MPI Communication between GPU Devices

We first evaluate the performance of our GPU DMA-assisted peer-to-peer intranode communication. In this test, we compare it against our previous design (shm), the shared memory based data transfer approach. The latency and bandwidth test both involve two processes, using both source and destination buffers in GPU memory. On Keeneland, we use GPUs 1 and 2, connected on the same IO hub (near case). On Magellan, GPUs 0 and 1 are connected to two different IO hubs (far case). All our experiments in this section evaluate large message transfers, i.e. for messages larger than 64 KB in our current setting. We also always pin the CPU controlling process to the socket that is closest to the controlled GPU. Figure 5.2 and Figure 5.3 compares the performance between these two cases.

First, we see that the DMA-assisted data communication is better than shm when two near GPU devices are used, primarily because DMA-assisted LMT avoids data staging through the host-side shared memory buffer. But when two GPUs are attached to different IO hubs, as in the far case, the data will also have to move through the IO hubs and the intersocket interconnects. Thus, we do not see any benefit of avoiding the memory staging via shared memory. On the other hand, the shm method uses both the sender and receiver to read and write data into the staging buffer, which could eliminate the message conflicts on the interconnect.

Second, comparing different LMT modes, COOPERATIVE mode is never the best. This is unexpected to us beforehand, since for the cooperative mode, we split the data into two halves, and let each GPU DMA engine work on one half. However the results shows that it loses at least to one of the other two modes in practice, which is possibly due to the interference between two GPU devices: when a peer direct access happens, the DMA engine will talk to some remote agent on the card device for data location translation and memory module commands issuing; therefore, when two DMA engines are working simultaneously, they could lead to contention.
Figure 5.2: MPI intranode latency/bandwidth comparison between using DMA-assisted intranode communication between peer-GPUs – in different LMT modes (“coop”, “get”, “put”) – and using Nemesis’ shared memory protocol (“shm”). Near case on Keeneland.

Figure 5.3: MPI intranode latency/bandwidth comparison between using DMA-assisted intranode communication between peer-GPUs – in different LMT modes (“coop”, “get”, “put”) – and using Nemesis’ shared memory protocol (“shm”). Far case on Magellan.
Figure 5.4: MPI intranode latency/bandwidth comparison between using DMA-assisted peer-to-peer communication in different LMT modes ("coop", "get", "put") and using shared memory ("shm"). Sharing GPU on Keeneland.

We further experiment with the case that two processes are sharing one GPU device, which is possible because both clusters have much more CPU cores than GPU devices. Figure 5.4 shows the result on Keeneland, and the result from Magellan is similar. The DMA-assisted data transfer benefits from fast data moving within a GPU device, resulting in an order of magnitude speedup over shm.

5.4.3 DMA-assisted Intranode MPI Communication between Host and Device

The DMA-assisted communication can work between a CPU process and a GPU process as well: exactly, the CPU process has a user buffer in its main memory, and the GPU process has the buffer in the device. Figure 5.5 shows such a result on Keeneland, that two processes are both pinned to the NUMA node 1, where GPU 1 and 2 are connected. We experiment with the two cases whether the CPU process is using the other devices connected to the same IO Hub, so that we can see the result of attaching and relaying, as explained in section 5.3.3.

The result here shows that the DMA-assisted LMT loses to shm. And a similar result is found on Magellan for two mixed processes pinned on further NUMA nodes. Comparing “attach” with “shm”, both data paths go from the GPU device to the memory node; the difference is only that in “shm”, data can be copied to the shared memory buffer without considering peer accessibility, while “attaching” case need start a new context there to access the exported memory. The implication is that the overhead of establish peer accessibility overpasses the benefit.
5.4.4 Application Evaluation: Stencil2D

Stencil2D from SHOC benchmark suite [20] measures the performance of a nine-point, two-dimensional stencil computation. It performs an iterative stencil computation on the GPU and requires a data exchange every \( \text{haloWidth} \) iterations. In this type of computation, processes are arranged in an \( N \)-dimensional Cartesian grid, and each process is assigned a corresponding section of a \( N \)-d array. Periodically, a process must obtain the values that its neighbors have calculated for the array elements that neighbor its patch, or its halo. Thus, this communication idiom, which is common across a broad range of iterative solvers, is referred to as a halo exchange.

Figure 5.6 shows the performance comparison against the original shared memory buffer based \texttt{shm} approach, using single and double-precision float data types, on Keeneland. In general
Figure 5.6: Stencil2D performance improvement on Keeneland, comparing using DMA-assisted peer-to-peer communication against using shared memory buffer based communication, with varying number of MPI processes.

This shows an average 4.7% speedup for single-precision and 2.3% for double-precision, by applying our system for accelerating the communication in stencil2D. Considering fewer double-precision units exist on GPUs than single-precision ones, communication takes a smaller portion, and gives smaller benefit for double-precisions.

In this workload MPI ranks are assigned to GPU devices in a round-robin manner. This explains the best benefit seen in the case of 4 processes, where rank 0 and rank 3 are assigned to the same GPU, and rank1 and rank 2 on the other two: the halo exchange happens first vertically, and then horizontally; in each step, DMA-assisted peer communication happens to apply between a pair and the original shm applied to the other pair. As a result, two sub-parts overlap in a mutually beneficial pattern; this overlapping can be somewhat seen in the 6-process case, but in a more complicated case as more MPI ranks sharing a GPU leads to a higher level of contention. The case of 2 processes is however quite unexpected to us. This may be due to the PCIe bus contention, since both parties are trying to send and receive equal-size messages; we plan to continue study this case as a future work.

Grouped according to problem sizes, 2048 vs. 4096 vs. 8192 (the matrix dimensions in the results are changed as the need to vary halo width), average speedup are, respectively, 8.5%, 3.9% and 1.6% for single precision, and 4.3%, 2.2% and 0.3% for double precision. One should notice that the computation increases quadratically with the problem size, and hence leaving smaller space for communication optimization. So is the case when halo widths are increased within each group.
5.5 Concluding Remarks

In this work, we explore the usage of GPU inter-process communication to design a DMA-assisted peer-to-peer data communication scheme for the intranode subsystem of a GPU-aware MPI. We present the design and discuss the challenges we face to integrate this scheme into MPICH. Further, we carefully studied the performance implication when communication parties reside at different places of a NUMA system. The augmented MPICH-GPU system is evaluated through both microbenchmarks and a halo exchange benchmark. Our microbenchmark evaluation shows the DMA-assisted peer-to-peer data communication is much beneficial when applied to two close GPU devices. An average speedup of 4.7% and 2.3% is observed on a stencil computational workload for single- and double-precision respectively, as compared against using the original shared memory based communication.
Chapter 6

RSVM: a Region-based Software Virtual Memory for GPU

6.1 Introduction

Different from the work in previous chapters, which designs transparent memory management subsystems behind MapReduce or MPI programming interfaces, the work in this chapter is designed as a general virtual memory system for a heterogeneous system.

As today’s OS virtual memory cannot manage the GPU memory space, we envision that a unified virtual memory systems that can abstract the separated memory domains across the PCIe bus will arrive. Recent initiatives in building similar systems as hardware-software co-designs, for discrete x86 co-processors [58] and GPU-integrated CPUs [9], have shown this trend.

A unified CPU-GPU virtual memory should support several important features missing in existing CPU-GPU memory management systems:

- It should present a unified address space to relieve programmers from manual CPU-GPU data movement.
- It should provide GPU memory management similar to the host-side paging mechanism.
- It should enable on-demand data fetching between the CPU and GPU memory spaces.
- It should provide a memory swapping mechanism for GPUs, giving a GPU task the illusion of a much larger device memory space.

In this chapter, we present Region-based Software Virtual Memory, a CPU-GPU memory management approach, to explore the GPU virtual memory design space. We propose using
regions as the basic data unit abstraction, which are defined by users and managed by RSVM for PCIe data movement and intra-kernel fine-grained GPU memory management. This design enables on-demand data fetching initiated from GPU kernels and a transparent swap mechanism backed by the host main memory. Meanwhile, an optimization space is still open to users through adjusting their region definitions.

This chapter makes the following contributions:

- We describe the design of a software virtual memory on a GPU-powered heterogeneous system, and demonstrate its novel features desired for GPU programming.

- We propose using user-defined regions as memory managing abstractions for such systems, and show that such abstraction leaves users with a useful optimization space.

- We introduce several new GPU system software mechanisms to serve as RSVM’s building blocks, including distributed region table management, CPU assisted swapping, and a software region TLB in the GPU non-cache-coherent shared memory.

- We provide a proof-of-concept implementation of RSVM on systems with x86 CPUs and Nvidia CUDA discrete GPUs.

- We conduct an evaluation with real GPU benchmarks to evaluate RSVM and identify certain missing architecture features desired for efficient GPU system software.

6.2 Background

Problematic manual GPU memory management: Figure 6.1 illustrates our target problem with a dense matrix multiplication implementation. The code consists of 4 steps: (1) GPU memory allocation, (2) input data transfer into the GPU memory, (3) kernel launch, and (4) result data transfer back to the host memory.

Several problems exist in the implementation. First, it assumes that all GPU buffers can be allocated, which holds true when the problem size is smaller enough and the available GPU memory resources are known beforehand. Second, manual PCIe data transfer brings coding overhead when performance optimization is adopted, as programmers must hard code for every different data transfer plan. Third, the static GPU memory allocation creates a larger memory working set for the GPU kernel than what the active threads actually need. As only a portion of all GPU threads are running at a given time, only the shaded sub-areas in the three buffers will be accessed with column-major thread scheduling.
6.3 Region-based CPU-GPU memory management

In this section, we give an overview of our proposed virtual memory system for heterogeneous computing, called Region-based Software Virtual Memory (RSVM).

6.3.1 Region as Memory Management Unit

CPU-GPU memory management must first decide a basic unit for GPU memory management and PCIe data transfer. Pages, typically used in CPU-based virtual memory, are often not the best option here. GPU threads are logically grouped into a grid of thread blocks, which are scheduled to run on a number of GPU SMs successively, following an order defined by the hardware scheduler. Therefore, using pages as the memory management unit may incur space inefficiency depending on hardware-specific thread block schedules. For example, consider the dense matrix multiplication program given in Figure 6.1. In cases where the hardware scheduler launches them in a column-major order, the working set for the active thread blocks is the shaded area in the three buffers. Pages, essentially fixed-length linear data blocks, increases the working set in B and C, as GPU threads touch every page for strided accesses. In addition, the PCIe bus, with much lower throughput than memories at either side, also suffers from inefficient page transfers for such strided accesses.

Similarly, it is difficult to enforce a fixed, one-size-fits-all memory management unit for different applications. In this work, we propose the use of region [27, 36], a user-defined data block, as the basic memory management unit. Each region is either an arbitrarily sized, contiguous area in memory (1-D linear region), or a group of such contiguous areas of identical
Figure 6.2: Dense matrix multiplication ($A \times B = C$) region programming example and running behavior

size and identical relative displacement (*stride*) in linear memory address (multi-dimension region). Currently, GPGPU APIs such as CUDA support linear, 2-D, and 3-D data layouts. In our proof-of-concept implementation and discussion in this chapter, we focus on 1-D and 2-D cases due to their rather dominant adoption in GPU applications, while the RSVM design can accommodate regions of 3 or more dimensions.

When a region is created, one specifies its extents in each dimension, as well as the stride. In the case of 2-D regions, the extents in x and y dimensions are also called *width* and *height*. Each region is represented by a system-wide unique ID, which is generated at the creation time. A host-side maintained heap, sitting atop RSVM, creates a region with `rsvm_malloc`. The region can then be used through a map operation, which translates the region ID into an address that can be used as a standard pointer.

In the dense matrix multiplication example, let $A$, $B$, and $C$ all be $N \times N$ single-precision float matrices. $A$ can be defined as a single 1-D region with $\langle \text{width, height, stride} \rangle$ of $\langle 8N^2, 1, 0 \rangle$. $B$ and $C$ are multiple 2-D regions defined as $\langle 8C, N, 8N \rangle$, where $C$ is the number of columns each thread block is assigned to compute (Figure 6.2). This way, the GPU working set for active thread blocks can be greatly reduced. The `unmap` interface marks the end of using a region in the GPU kernel code, so a region whose processing has completed can be removed from the working set. Note that multiple fine-grained regions are grouped into a region collection (*rgn_coll*) in the sample code, whose metadata (*rgn_coll_meta*) contains all the associated regions’ IDs. Section 6.4.1 gives more details on RSVM APIs.
6.3.2 Benefit of Region-based Approach

Adopting regions have multiple benefits. First, regions hide non-continuous memory domains' management from developers: users need not care where a region is, but simply focus on creating and using it. Second, users can still optimize the granularity of memory access and PCIe data transfer, by adjusting the region definition taking into consideration of data distribution among GPU threads. Co-scheduled thread blocks can give runtime information regarding the working set to reduce GPU memory usage. Such optimization need to consider the tradeoff in region sizes according to individual application's needs: large regions yield better PCIe transfer performance, while small regions reduces both wasted device memory space and unnecessary data transfer. Finally, assuming regions are correctly defined, the underlying system can leverage the working set information for system functions that cannot be done today, e.g. local memory domain buffering, on-demand data fetching, and transparent data swapping.

**Unified address space backed by regions** To some extent, the region ID space presents a unified address space across segmented CPU and GPU memory spaces. Such a unified address space is different from the current virtual address space, such as CUDA Unified Address Space (UVA) \[5\], which maps GPU memory into a distinctive segment in virtual memory address space and still requires explicit copying of data across CPU and GPU. In contrast, such copying is unnecessary for RSVM. Furthermore, pointer-based data structures must be marshalled across PCIe and reconstructed after.

Using this unified address space, Listing 6.1 shows a linked list prepared on the host side but consumed on the device. This feature is especially useful for CPU-GPU co-processing on shared data structures. Here an RSVM pointer consists of a region ID and an offset within the region. A host-side heap maintains a group of internal buffers, each of which, as a RSVM region, contains a configurable number of free elements for linked list nodes.

6.4 RSVM Design

In this section, we describe the design of our prototype region-based CPU-GPU memory management system, focusing on addressing several unique challenges:

- **Parallellism:** traditional system design did not consider the huge paralellism possessed by GPUs.

- **Lack of synchronization tools:** traditional software techniques, often utilizing synchronization methods like locks and semaphores, do not suit the throughput-oriented processor, where a large number of hardware threads are over-subscribed for latency hiding.
Listing 6.1: A linked list in the unified address space

typedef struct {
    rsvm_id id;
    unsigned long long offset;
} rsvm_ptr_t; // a RSVM unified space pointer
typedef struct {
    type value;
    rsvm_ptr_t next;
} node_t; // a linked list node
// host code: host-side heap on RSVM,
// maintaining free elements in buf[n]
rsvm_malloc() {
    //... when no free element can be found
    buf[n++] = rgn_create_cpu(sizeof_internal_buffer);
}
// host code: create a linked list
rsvm_ptr_t head = rsvm_malloc();
node_t *n = rgn_map_cpu(head.id,
    op_writeonly, NULL, NULL) + head.offset;
n->value = value;
n->next = rsvm_malloc();
//... to add more nodes

// device code: traverse the linked list
traverse(rsvm_ptr_t head) {
    rsvm_ptr_t p = head;
    while (p != NULL) {
        node_t *n = rgn_map_gpu(p.id,
            op_readonly, &complete, &req) + p.offset;
        if (!complete) n = rgn_wait_map_gpu(
            p.id, req) + p.offset;
        //... to visit
        rsvm_ptr_t r = n->next;
        rgn_unmap_gpu(p.id);
        p = r;
    }
}
Distributedness and asynchrony: memory access and management span both the GPU and the CPU sides, where computation proceed asynchronously.

Obscurity and diversity: GPUs and the vendor drivers remain closed, with differences across vendors and generations. This requires the system assume the least on the GPU architecture and system support.

The architecture of RSVM is depicted in Figure 6.3, which shows its placement and internal software hierarchy. RSVM is built on top of the GPGPU APIs provided by either open-source or proprietary drivers. Below we describe the major components of RSVM.

6.4.1 Region API

GPU parallelism is created by the concurrent execution of multiple SIMD engines, each running warps of threads in lock steps. RSVM requires that threads within a warp invoke the same API function with the same parameters synchronously. Although warp specialization is possible, differentiating threads within a warp leads to SIMD lane divergence and is typically discouraged due to performance concern.

Table 6.1 summarizes the RSVM APIs. Most functions have both CPU and GPU versions, to be called in host/device codes. Among them, rgn_map does ID-to-pointer translation. Based on an op (readonly, readwrite, or writeonly) code, RSVM knows how users code access a region. It returns asynchronously even when a region’s data copying from the remote side is incomplete. In this case, the complete flag is reset and a request number is assigned. Instead of busy waiting, the program can perform other useful work before calling the blocking wait API, which returns upon the completion of data transfer. Note that a multi-dimensional region can be created only at the host side, as such more complex data structures are typically produced there and consumed by the GPUs. When it is mapped at the GPU, its stride might be modified for GPU
Table 6.1: Region APIs and region collection utility functions

<table>
<thead>
<tr>
<th>API</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>rgn_create_cpu/gpu(size)</td>
<td>create a linear region locally (either on the host or GPU)</td>
</tr>
<tr>
<td>rgn_create_cpu(size, addr, stride, width, height)</td>
<td>create a multi-dim region on the host</td>
</tr>
<tr>
<td>rgn_free_cpu/gpu(rgn)</td>
<td>delete a region</td>
</tr>
<tr>
<td>rgn_map_cpu/gpu(rgn, op, *complete, *req)</td>
<td>map a region locally</td>
</tr>
<tr>
<td>rgn_map_gpu(rgn, op, *complete, *req, *pitch)</td>
<td>map a multi-dim region on GPU, find the local pitch</td>
</tr>
<tr>
<td>rgn_wait_map_cpu/gpu(rgn, req)</td>
<td>wait until a mapping request is finished</td>
</tr>
<tr>
<td>rgn_unmap_gpu(rgn)</td>
<td>unmap a region on GPU</td>
</tr>
<tr>
<td>rsvm_sync()</td>
<td>sync regions information at a synchronization point</td>
</tr>
<tr>
<td>rgn_coll_create_cpu(collect_sz, n_rgn, rgn_size)</td>
<td>create a collection of linear regions</td>
</tr>
<tr>
<td>rgn_coll_create_cpu(collect_sz, n_rgn, rgn_size, stride, width, height)</td>
<td>create a collection of multi-dim regions</td>
</tr>
<tr>
<td>rgn_coll_get_meta_cpu/gpu(coll_id)</td>
<td>map the IDs in the collection to a local memory buffer</td>
</tr>
</tbody>
</table>

memory’s alignment and returned in \( \text{pitch} \). \text{unmap}, available as a device-side API only, marks the end of using a region on the GPU. Together with \text{map}, they form a memory access session in the GPU code, in order to let RSVM know how GPU code in the session accesses the region. RSVM relies on users to correctly pair \text{map} and \text{unmap}. This is unnecessary at the host side, as the OS virtual memory understands the working set of the process through paging.

The RSVM synchronization \text{rsvm_sync}, a host-side function, lets the asynchronous runtimes on the CPU and the GPU exchange their knowledge of updates to region states since the previous synchronization. Data updates across the PCIe bus are delayed to subsequent mapping operations. \text{rsvm_sync} can be invoked before a GPU kernel or after a synchronized GPU command, therefore regions will not experience a \text{rsvm_sync} event during a kernel function’s execution.

Table 6.1 also shows RSVM’s region collection-related utility functions. For example, \text{rgn_coll_get_meta} returns IDs of all the regions in the collection, in a local memory buffer.

### 6.4.2 Region States

A region’s state defines its current status, while the RSVM state transition protocol defines the state transition rules. RSVM uses an MSI-like protocol [29] as shown in Figure 6.4. Here, a modified region is one that has been modified since the last synchronization. A shared region has a clean local copy. An invalid region has no or obsolete data. In addition to the above three common states, the GPU side has two more, namely modifying and sharing. They have similar
status with the *modified* and *shared* states, but are identified as “currently in use” (by a pair of *map* and *unmap* operations). This allows the GPU-side *swap* and *reclaim* operations (see Section 6.4.5) to flush unmapped dirty data to host memory and release GPU-side buffers (along with region state changes).

### 6.4.3 Region Management

Beside a region’s state, RSVM internally stores its metadata, e.g. addresses in each memory domain, dimensions, in a *region table*, indexed by the ID. It has a copy on each side, with local changes merged at *rsvm_sync* operation. The region table, as well as the ID space, is statically partitioned into fixed-length *segments*. A segment is *owned* by either the CPU or the GPU at a given time, allowing only one side to create regions in its ID space. When a new region is to be created, RSVM finds an unused entry in a segment owned by the local side, which ensures a unique region ID without inquiring the other side. Obviously, if the region is deleted by the non-owner side, its region table entry will not be recycled by the owner till the next synchronization point.

A *region manager* sits on each side and serves user requests. It queries the region table for region states and a lower-level memory allocator to manage the local memory domain. Region managers communicate with each other for CPU-GPU cooperative tasks. When a region is created or mapped for the first time, the region manager creates a buffer for it in the local memory domain. If an *invalid* region is mapped, the region manager fetches its up-to-date copy into the local buffer. On the GPU, the region manager itself cannot initiate a PCIe data transfer and must communicate with the host side service through a GPU callback (Section 6.5.1). Upon freeing a region, the manager returns the buffer to the memory allocator.
6.4.4 CPU-GPU Region Consistency

As the CPU and GPU work asynchronously, it is possible that one side is unaware of the modification to a shared region on the other. Not designed to invent new synchronization methods or consistency models, RSVM adopts a relaxed consistency model. This is based on the consideration that a strong consistency model would demand frequent synchronization across the PCIe bus, while most CPU-GPU co-processing applications have rather coarse-grained data/task distribution. As a result, RSVM reports the conflict when concurrent data modification is identified at runtime.

6.4.5 Enabling Swap for GPU Memory

unmap does not free memory resources immediately. RSVM devises a swap mechanism in the GPU region manager, using the host main memory as the backing store. Such swap operations can happen within a single large kernel, or after multiple kernel runs.

RSVM's swap design considers two challenges. First, as GPU thread scheduling cannot be controlled, it is impossible to keep any daemon GPU threads which will wake up and are only responsible for swap. Second, region replacement policy requires online region access accounting and a victim algorithm to find candidate regions. Both can be substantial overhead with GPU’s massive parallelism.

As no GPU thread is dedicated for swapping, RSVM's swap starting/stopping logic is embedded in map and unmap paths. However, instead of hijacking a GPU thread to complete a swap operation, the swap logic is designed asynchronous and re-entrant, and aided by a host-side assistance service. Swap and region reclamation are separated as two independent steps.

For region access accounting, RSVM keeps a Not-Frequent-Used (NFU) counter for each region in the GPU memory. It is incremented in each map operation. When the GPU memory availability is low, a GPU thread requests the host-side service thread to transfer the counters buffer--and needed segments of the GPU’s region table--back into the host and start a traditional counter-based replacement algorithm, e.g. the aging algorithm RSVM currently employs. When the candidate regions are chosen, the host service starts PCIe data copies to swap those dirty ones out to the main memory. The host-side service does not modify the GPU-side region table, and does not block the signaling GPU threads.

As the swap is happening on the CPU, other GPUs thread may re-enter the swap logic in the GPU. Seeing an ongoing request, they simply leave. When a thread finds the previous swap is done, it will see the swapped regions' IDs returned from the last request, as well as the NFU sorted list. Among them those regions which have not been updated since last call will be set to the shared state, and become potential candidates for region reclamation.

Region reclamation happens as another independent operation of the GPU region manager.
It simply reclaims the memory buffers for shared regions. It honors the latest NFU result, and changes the state of reclaimed regions to invalid.

6.4.6 A Software TLB for Region Mapping on GPU

Modern CPUs have a Translation Lookaside Buffers (TLB) cache for efficient converting virtual to physical memory addresses. As RSVM’s GPU runtime stores the region table in the device memory, it employs a software TLB mechanism using the fast on-chip shared memory inside each SM. As the shared memory is a thread-block-specific manually operated scratch-pad space, RSVM’s software TLB enforces coherence between the shared memory and the GPU device memory, and between shared memory spaces of two different SMs.

**Coherence.** Between the region table in the device memory and the cached portion in the shared memory, RSVM software TLB adopts a write-through policy.

To enforce coherence between two TLBs in different thread blocks, RSVM’s solution is that, when reading from a TLB entry, the threads distinguish safe and unsafe states. A region is safe, if its cached status is mapping or sharing states already. These states indicate that some other threads in this thread block are using it, which implies two facts: those threads have cached the region table entry, and that the TLB cached entry is updated, as the region is still in use, and no threads will try to modify its metadata. Otherwise, the thread proactively reads the region table entry to refresh the TLB entry and sends it to the safe state.

To coordinate the TLB entry state updating between thread warps, a reference counter is used in the shared memory. It is updated atomically in map and unmap operations. The first warp trying to use a region, i.e. the one successfully updates the counter value from 0 to 1, updates the global region entry (e.g. possibly changing the global region state) and prepares the TLB entry. Then the TLB entry is safe, and the rest threads need not touch the device memory until the last one, which will try to update the global state.

**TLB design.** This software TLB implements a fully-associative policy, and enjoys a thread warp’s parallelism for look-up. Each entry has a status flag to indicate if the entry is mapped to a region or unused. A thread block-wise lock is used to protect changes to any entry’s status flag. After that, the entry’s consistency is maintained through atomically updating the reference counter and its state of the entry. When the reference count of a TLB entry is zero, it can be reused for caching another region table entry. RSVM currently do not provide a TLB overflow evicting scheme, as that would violate the safeness mechanism above. As we leave the software TLB size as a configurable parameter, we assume that the user is able to estimate the maximum number of concurrently used regions by a thread block.
6.5 Implementation

This section will cover several important building blocks of RSVM.

6.5.1 GPU Callback

GPU off-the-shelf systems today have no such a mechanism to let GPU code to invoke a host-side CPU function call. Yet RSVM’s GPU-side runtime need occasionally make such functions, e.g. when the GPU-side region manager tries to fetch the latest data for an invalid region. A software mechanism, called GPU callback, exists as a workaround, which has a CPU thread polling on a flag variable which is shared with a GPU [65].

In addition to the existing GPU callback mechanism, RSVM implements a new type of collective callbacks, for the case that many GPU threads ask for the same request. The GPU side runtime detects all concurrent GPU threads for the same callback, and sends only one single signal across. Upon completion, it lets one thread take care of finishing the task. Collective callbacks avoid the traffic surge on the PCIe bus and save the CPU from waisting repeated effort.

RSVM implements three types of GPU callbacks:

- **Handling a region fault**: non-collective, asynchronous, and parameterized callback. It asks for transmitting the data of an invalid region, and returns asynchronously a request number as well as a complete tag.

- **Getting a new region segment**: collective, synchronous, and non-parameterized callback. It asks a new segment in the region table. It blocks all calling threads and sends a single signal to the host. It returns for all of them when a new segment is allocated to the GPU side.

- **Starting the swap**: collective, asynchronous, re-entrant, and non-parameterized callback. It asks the swapping service on the host side to fetch modified regions to the main memory. It returns asynchronously with a complete tag, which lets one thread to finish up on host-side completion.

6.5.2 Lower-level Memory Allocator on GPU

Similar to the host-side virtual memory system, a way of managing the underlying GPU physical memory resource is needed. RSVM relies on ScatterAlloc [64], a memory heap on the GPU. When RSVM is started, it is created with a predefined size, which limits the total GPU memory usage for a process. The host-side underlying memory management is reliant on a normal main memory heap.
Table 6.2: RSVM vs native in benchmarks that fit in the device memory (BackProp has two kernels)

<table>
<thead>
<tr>
<th>Workload</th>
<th>Problem size</th>
<th>Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>MatrixMultiplication</td>
<td>8192x8192x8192 (768 MB)</td>
<td>21%</td>
</tr>
<tr>
<td>BlackScholes</td>
<td>33554432 (64 MB):32 iterations</td>
<td>5%</td>
</tr>
<tr>
<td>HotSpot</td>
<td>1024x1024 (12MB):16 iterations</td>
<td>1%</td>
</tr>
<tr>
<td>BackProp</td>
<td>262144 (K1:19.93MB/K2:36.71MB)</td>
<td>5%</td>
</tr>
</tbody>
</table>

6.5.3 Limitation of current implementation

RSVM is currently implemented as a user-level library, limiting its use in a single GPU application process. In the case of several host processes sharing the same GPU device, currently RSVM cannot dynamically adjust the GPU memory usage across processes as it lacks a global view. This can be extended by sharing the region table across multiple processes, utilizing shared memory mapping on the host side and cudaIPC shared memory functions on the device. RSVM is implemented currently for a single GPU device, but, in principle, can be extended to the multi-GPU scenario. Finally, users must launch their GPU kernels in a CUDA stream, which is required to overlap a GPU kernel with PCIe data transfers.

6.6 Evaluation

In this section, we evaluate RSVM across a set of GPU benchmarks. Our testbed is an Intel x86 system with Xeon E5507 CPU, 6 GB main memory, and an Nvidia GTX480 Fermi GPU card (15 streaming multiprocessors and 1.5 GB device memory). The software is Ubuntu 10.04LTS Server, with an x86_64 Linux kernel (2.6.32-33), and CUDA 5.0rc.

6.6.1 Benchmark

First, we ported a group of benchmarks from Nvidia CUDA SDK [5] and Rodinia benchmark suite [8] to RSVM. These workloads all have problem sizes that can fit into our GPU card’s device memory. Among them, matrix multiplication and BlackScholes are relatively computational intensive and HotSpot and BackProp are relatively memory intensive. When we define regions, we choose to take individual buffers as regions for HotSpot and BackProp as their buffers are small enough. For BlackSchole and MatrixMul, we define multiple fine-grained regions for each buffers. All experiments are repeated three times and taken average, and timing includes both computation and PCIe transfer time but not disk I/O or input generation time (the same for all experiments through this section).

Porting results are compared with their original native implementations (Table 6.2). Three workloads, except for MatriMul, shows RSVM, as an extra software layer, introduces
small overhead. However, for MatrixMul the slowdown is 21%. An examination reveals that RSVM brings a pressure on GPU register file, increasing the register usage count from 25 up to 60 (per GPU thread), as a few hundred lines of code from RSVM’s GPU device library, as well as ScatterAlloc, are compiled into the GPU kernel. Also note that with ScatterAlloc compiled in, register count is already 32. Required by GPU hardware threading, register files are statically partitioned to each GPU thread. The increase in register usage decreases the streaming multiprocessor occupancy, i.e. the simultaneous active threads per SM is decreased from 1024 in the native version to 512. MatrixMul, a computational intensive workload, has no other resource bottleneck on GPUs as for other three workloads. Thus a decrease in occupancy shows rather large slowdown.

6.6.2 Matrix multiplication with increasing problem size

In this experiment, we use matrix multiplication to evaluate RSVM when problem sizes exceeds the GPU device memory limit. Although MatixMul shows slowdown on RSVM in the previous example, we still choose it, because it is typically used to evaluate hardware computational power (GFLOPS).

As the problem size increases beyond the GPU memory limit, the native version will not run. For performance comparison, we implement a manual version. It partitions matrix B and C into multiple column-wise sub-matrices, and creates buffers in GPU memory that can store matrix A and a partition from matrix B and C. It takes iterations to upload each partition to the GPU memory, run the computation kernel, and download the result partition to C.
The RSVM version uses the same implementation with same parameter for all input sizes. As this matrix multiplication turns out to have a column-major scheduling for launching thread blocks, we choose the region definition as in Figure 6.2. Matrix A is created as a single region. Matrix B and C are created as 16 floats wide 2-d regions, with their heights and strides as the height and the width of Matrix B and C, respectively.

The result is shown in Figure 6.5. First, note that the GPU memory size is 1.5 GB, but, in RSVM, we can use only 1280 MB at most for the managed GPU memory. Although our RSVM library uses only a few dozen MB internally, it turns out that the GPU reserves some memory for its own use, and CUDA’s GPU `cudaMalloc` allocator has probably fragmentation issues, which are only shown in creating various different GPU buffers inside RSVM. Due to the cubic computation complexity, the manual version attains a saturated GFLOPS. RSVM version, despite occupancy problem, almost steadily reaches 70% of the saturated throughput, although the swap mechanism starts to kick in at 1200 MB. After all, RSVM survives the large problem size, requiring no code change, and its swap mechanism shows less than 10% overhead.

Alternatively, we test a wrongful region definition, by defining multiple linear regions for matrix A and C and a single region for matrix B. While it can run at 768 MB, with an 25% overhead below the correct RSVM version, thrashing occurs at 1200 MB. This shows that it is better to leave choices of region definition to application developers.

### 6.6.3 Breadth-first search

The third experiment is a breadth-first search (BFS) graph algorithm. Graph algorithms are known for dynamic memory access patterns, which depend on input graphs and are hard to predict before running. Therefore, it is hard for programmers to find an optimal schedule as with MatrixMul. Meanwhile, BFS is a memory intensive workload, which we use to evaluate RSVM in contrast to computation-intensive MatrixMul.

The BFS native implementation is from SHOC benchmark [21]. It implements a warp-centric algorithm [31], by parallelizing vertices among different warps, and the edges adjacent to the same vertex among the threads of that warp. The graph is prepared in an adjacent list format with an array of edges and an index array for each vertex into the edge array. To port it to RSVM, we cannot know how threads are accessing the data upfront, and thus choose to define two arrays each as a collection of same-sized linear regions—we leave the region size as adjustable parameters. Another cost array for each vertex to record its distance from the source is created as a single large region.

The input graphs we prepared are from the 10th DIMACS Challenge [1] except for `rmat` and `rmat-g500`, which are generated from GTgraph [3]. Of a graph with $n$ vertices and $m$ edges, the edge factor $\frac{m}{n}$ gives the information of the graph’s structure. It is also related to BFS iterations,
or depth. In Table 6.3, the upper 5 graphs (“small”) have problem sizes that can fit into our GPU memory limit; the lower 4 graphs (“large”) are chosen to exceed the limit of 1280 MB.

The algorithm’s performance are measured in traversed edges per second, or TEPS.

**Small graphs.** For the small group (Figure 6.6), RSVM exhibits slowdown for those graphs this implementation is not good at, which typically have a low edge factor and a long depth. The performance improves as the edge factor increases.

As BFS has a dynamic memory access pattern, RSVM adds to each vertex visiting an overhead of mapping regions for its adjacent vertices that a thread warp is going to access. This overhead is amortized with a large edge factor. But when the depth is long, for each region mapping, few vertices are actually used. This explains the improved performance when the edge factor increases and the depth decreases. For the last one, a GRAPH500 [2] input graph, RSVM shows 4% improvement over the native implementation, due to amortized overhead and overlapping of GPU kernel and PCIe data communication.

Table 6.3: Benchmark graphs (n and m in $10^6$). Small (upper, fit GPU memory limit) and large (lower, exceed GPU memory limit). “kernel”: launching configuration as “number of blocks/block size”.

<table>
<thead>
<tr>
<th>Name</th>
<th>n</th>
<th>m</th>
<th>m/n</th>
<th>depth</th>
<th>kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>germany.osm</td>
<td>11.5</td>
<td>24.7</td>
<td>2.14</td>
<td>4620</td>
<td>89/256</td>
</tr>
<tr>
<td>nlpkkt160</td>
<td>8.35</td>
<td>221.2</td>
<td>26.5</td>
<td>163</td>
<td>128/256</td>
</tr>
<tr>
<td>in-2004</td>
<td>1.38</td>
<td>27.2</td>
<td>19.7</td>
<td>26</td>
<td>86/256</td>
</tr>
<tr>
<td>coPapersCiteseer</td>
<td>0.43</td>
<td>32.1</td>
<td>73.9</td>
<td>26</td>
<td>53/256</td>
</tr>
<tr>
<td>kron_g500-simple-logn21</td>
<td>2.10</td>
<td>182.1</td>
<td>86.8</td>
<td>0</td>
<td>128/256</td>
</tr>
<tr>
<td>nlpkkt200</td>
<td>16.2</td>
<td>432.0</td>
<td>26.6</td>
<td>203</td>
<td>248/256</td>
</tr>
<tr>
<td>arabic-2005</td>
<td>22.7</td>
<td>640.0</td>
<td>28.1</td>
<td>98</td>
<td>348/256</td>
</tr>
<tr>
<td>rmat</td>
<td>4.19</td>
<td>536.9</td>
<td>128</td>
<td>5</td>
<td>64/256</td>
</tr>
<tr>
<td>rmat-g500</td>
<td>4.19</td>
<td>536.9</td>
<td>128</td>
<td>5</td>
<td>64/256</td>
</tr>
</tbody>
</table>
Large graphs. For large graphs, BFS native version will not run. Therefore we implement two normal ways to deal with the case. 1) **Manual:** it partitions the graph into multiple parts, each fitting into the GPU memory. Then, it take turns to run BFS on each part for each BFS iteration. Data swapping is manually done across the host side buffers with the GPU memory partition buffers for each turn. 2) **UVA:** this method leaves the buffers on the host, and maps them into GPU’s address space so that GPU threads can access them directly, but through the PCIe bus. One should notice that optimizations can be taken onto them, e.g. double buffering for the manual approach to overlap computation with data communication. However, two above methods are fundamentally different, with rather distinctive code bases to maintain.

Figure 6.7 shows their performance methods compared with RSVM, across four different graphs with similar sizes (1.7 GB - 2.5 GB). Between manual and UVA approaches one wins over the other depending on graph structures. For graphs with small edge factors and long depths, the manual approach performs worse, as excessive data is constantly swapping into and out of GPU each turn, with very small portion used actually, while the UVA version is good at accessing only the needed data over the PCIe bus. For graphs with large edge factors and low depths, the manual approach performs better as it transmits a partition at a time into the GPU memory, benefiting from PCIe efficiency and local data accesses, whereas in the UVA version simultaneously running threads access adjacency list data over the PCIe bus, making it a bottleneck.

RSVM enjoys the benefit of both, as region sizes can be adjusted, and are fetched into GPU memory on demand. The results show that RSVM meets the better performance between manual and UVA methods, and even surpasses them for two R-mat generated low-depth graphs.
6.6.4 Discussion

As shown in the experiment of MatrixMul, added register file pressure due to the RSVM library can hurt the performance very much. Static register file partitioning strategy counts the same number of registers to all threads. However, the RSVM library code is executed in only a small number of GPU threads at a time. Dynamically sharing registers between threads will be able to reduce such register file pressure.

Lack of a privilege mode on GPU leaves RSVM device library data structures under-protected. Misbehaving GPU threads in the user code can pollute RSVM’s data structure. This has led to debugging difficulty in our experience, and is also a security issue.

6.7 Concluding Remarks

GPU virtual memory provides both ease of programming and performance benefits. As of today, due to the lack of architectural support, there are no virtual memory systems that abstract/unify discrete host and GPU memory spaces. In this chapter, we present RSVM, our exploratory software virtual memory system and demonstrate its convenience in programming as well as performance improvement. Meanwhile, RSVM is shown to incur rather significant overhead in certain use cases, revealing the need for additional architectural features for future-generation GPUs to benefit more from software virtual memory management.
Chapter 7

Conclusion

In this PhD thesis study, we show two categories of software runtime support approaches in assisting memory management in GPU-enabled heterogeneous systems. Our GPU memory management subsystems within MapReduce and MPI show not only easier programming interfaces but also improved performance. Specifically, we abstract the GPU on-chip shared memory for the usages of transparent data buffering to accelerate MapReduce workloads on GPUs, and extend MPI for GPU-awareness to support GPU memory buffers, which further enables optimized intra-node communication through pipelining and using GPU DMA engines. Then in order to provide a generalized GPU memory management system, we propose a Region-based Software Virtual Memory system (RSVM) to hide the separated memory domains. RSVM not only takes over GPU memory management and CPU-GPU data movement, but also presents novel features, e.g. GPU-initiated on-demand fetching and transparent swap mechanism for GPU memory, which cannot be realized within current GPU system models.
REFERENCES


83


