Characterization and Exploitation of GPU Memory Systems

Kenneth S. Lee

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

Master of Science
in
Computer Science and Applications

July 6, 2012
Blacksburg, Virginia

Keywords: GPU, APU, GPGPU, Memory Systems, Performance Modeling, Data Transfer

Characterization and Exploitation of GPU Memory Systems

Kenneth S. Lee

(ABSTRACT)

Graphics Processing Units (GPUs) are workhorses of modern performance due to their ability to achieve massive speedups on parallel applications. The massive number of threads that can be run concurrently on these systems allow applications which have data-parallel computations to achieve better performance when compared to traditional CPU systems. However, the GPU is not perfect for all types of computation. The massively parallel SIMT architecture of the GPU can still be constraining in terms of achievable performance. GPU-based systems will typically only be able to achieve between 40%-60% of their peak performance. One of the major problems affecting this effeciency is the GPU memory system, which is tailored to the needs of graphics workloads instead of general-purpose computation.

This thesis intends to show the importance of memory optimizations for GPU systems. In particular, this work addresses problems of data transfer and global atomic memory contention. Using the novel AMD Fusion architecture, we gain overall performance improvements over discrete GPU systems for data-intensive applications.The fused architecture systems offer an interesting trade off by increasing data transfer rates at the cost of some raw computational power. We characterize the performance of different memory paths that are possible because of the shared memory space present on the fused architecture. In addition, we provide a theoretical model which can be used to correctly predict the comparative performance of memory movement techniques for a given data-intensive application and system. In terms of global atomic memory contention, we show improvements in scalability and performance for global synchronization primitives by avoiding contentious global atomic memory accesses. In general, this work shows the importance of understanding the memory system of the GPU architecture to achieve better application performance.

This work was supported in part by the NSF Center for High-Performance Reconfigurable Computing (CHREC).

To my Family and Friends

Acknowledgments

I would like to use this page to express my deepest gratitude to those that allowed me to achieve this point in my academic career.

First and foremost, I cannot say how grateful I am to Dr. Wu-chun Feng, my research advisor and committee chair. His tireless work ethic inspired me to work as much and as hard as I could to produce research of the highest quality. I think Dr. Feng for instilling confidence in my work as well as in myself. My two years under him have truly been memorable.

I would also like to thank Dr. Heshan Lin for providing me much needed guidance in the publication of my AMD Fusion work. It has been a pleasure collaborating with Dr. Lin on this work, which has made me a much better researcher. I would also like to thank him for being a part of my defense committee.

I am also thankful to Dr. Yong Cao for being on my research committee, as well as helping to educate me as to the state of the art in GPU computing.

Often, it was very stressful working many long hours with very tight deadlines. In those dark times I was glad to have Paul Sathre and Lee Nau to help keep my spirits high. I thank them both, from the bottom of my heart, for keeping me sane in those moments.

I would also like to thank SyNeRGy members for providing me with constructive criticism and research questions, which greatly improved the quality of my work.

I thank Gregory Gates for his tireless work as my best friend and compatriot. Gregory made my undergraduate CS classes fun and was always willing to play games with me if I needed a break from work. He was also the best roommate possible for my entire college experience spanning 4 years.

In addition to my friends inside of the CS community, I would also like to thank some of my friends that helped broaden my experiences here at Virginia Tech. Specifically, I would like to thank Charles Neas, Lera Brannan, Sarah DeVito, Mitchell Long, and Nora McGann, members of ΣAX.

I do not know how I could ever thank Lisa Anderson enough. She has been the source of my strength for my entire college career, and constantly inspires me to strive for greatness. As my girlfriend of six years she has given me tremendous support, comfort, and love as I worked to complete this degree.

Last, but certainly not least, I would like to thank my parents and family who of supported my academic endeavors for my entire life and put their utmost faith in me throughout. I could never have been here today without their tremendous support and care.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Graphics Processing Units (GPUs) are an exemplar of modern trends in high performance computing and supercomputing environments, in which massive parallelism through simpler, more energy-efficient processing units reign supreme over complex traditional CPU systems. Three of the top ten supercomputers in the world leverage GPU technology for the majority of their performance [1]. In addition to their use in supercomputers, GPUs have benefited scientific and high performance computation in desktop systems. Because of the ubiquity of GPUs in commodity systems, the use of General-Purpose Computation on Graphical Processing Units (GPGPU) gives even modest desktop systems a large amount of computational power.

The GPU has three distinct benefits over traditional homogeneous processing systems. Firstly, the immense computational power that is possible with a single GPU is much greater than is possible with a single CPU, or even multi-CPU systems. Figure 1.1 shows the com-

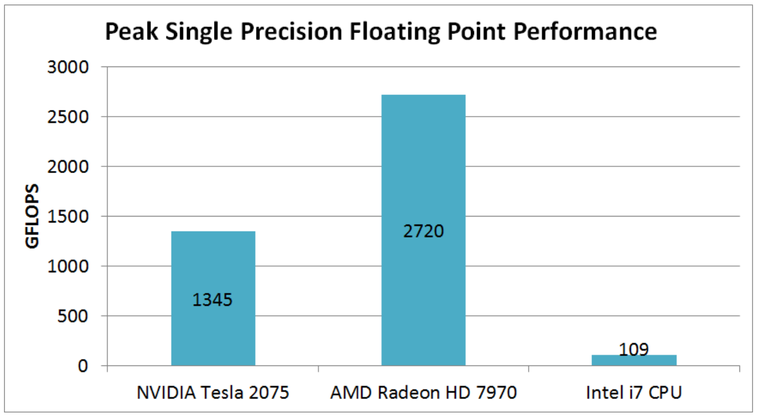


Figure 1.1: Comparison of Peak Single Precision Floating Point Performance

parative performance of both NVIDIA and AMD GPUs when compared to an Intel CPU device showing the performance discrepancy between CPUs and GPUs. Secondly, the energy efficiency of GPU systems, as noted by performance per watt achievable is traditionally better than CPU-based systems. Finally, the cost to performance ratio, partially driven by the size of the commodity GPU market, is far better than CPU systems. These benefits are achieved through the extensive use of the Single-Instruction Multiple-Data (SIMD) paradigm throughout the architecture of the GPU.

Although the GPU is capable of achieving tremendous performance, it is not a panacea for all programming woes. It is quite difficult, and for most realistic cases impossible, to write an application which leverages all of the raw compute power that the GPU can provide. Figure 1.2 shows the performance efficiency of the top 100 fastest supercomputers when running the LINPACK application. As the graph shows, the relative efficiency of accelerated systems is around 40%-60% for the majority of those systems. On the other hand, CPU-based supercomputers achieve upwards of 80% effeciency. There are many reasons that cause this relatively low efficiency for GPU-accelerated systems, including issues in the memory system, compute system, and the programming system. For this work we will focus our efforts on addressing the issues of the memory system, specifically addressing the problems of data

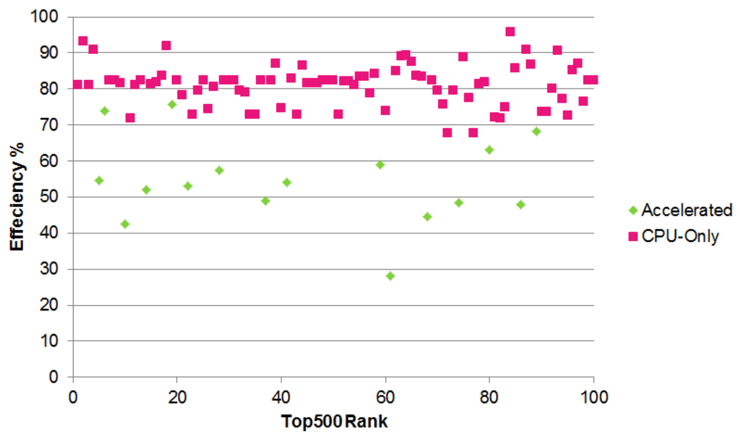


Figure 1.2: LINPACK Performance Efficiency of the Top 100 Fastest Supercomputers

transfers and global memory contention.

As an example of the impact of data transfer overhead on performance, we show the percentage of overall application time spent on the three stages of computation: data transfer to the GPU, data transfer from the GPU, and kernel execution for a vector addition algorithm. The results from this application based on these stages is shown in Figure 1.3. From this graph, we find that less than 5% of the overall application time is spent performing useful work, kernel execution. Over 95% of the application's execution is spent performing data transfers. Because of the large cost of data transfers, it often is not worth the effort to perform these types of *data-intensive* computations on the GPU.

The AMD Fusion is a novel architecture designed to eliminate the PCIe bottleneck. This architecture places the CPU and GPU on the same die, replacing the PCIe interconnect with an on-die, high-speed memory controller and providing a shared memory system between the CPU and GPU. This architecture therefore offers the opportunity to greatly reduce the costs of data transfers for GPGPU applications. However, memory access patterns on this new architecture can be quite complex, so an exploration and characterization of memory movement and data paths is required to fully optimize performance on this system.

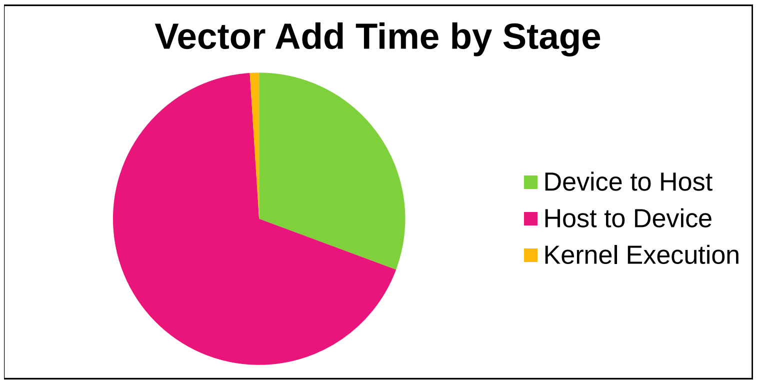


Figure 1.3: Percentage Time per Stage of Computation for the VectorAdd Application

In addition to improving out-of-core memory system performance, via improved data transfer, we also investigated the in-core problem of global memory contention on GPU memory. We look at an instantiation of this problem in terms of global synchronization primitives on GPU systems. Taking into account the problem of memory contention, we create two novel distributed synchronization primitives and compare them against existing approaches. By eliminating the overhead associated with this global memory contention, we are able to improve the overall performance of these primitives.

We show in this work that a better understanding of the memory system and data transfer characteristics of a given architecture can lead to the exploitation of that architecture to achieve better performance over traditional approaches. We describe the ideas of this work both for the AMD Fusion architecture, in terms of eliminating PCIe overhead, and for all GPU systems by achieving better performance for global synchronization primitives through the reduction of memory contention on that system.

1.2 Related Work

The use of GPUs to accelerate computation of applications has been well documented in the community of graphics [43, 16, 2], bioinformatics [31, 12, 5], and many other fields [33, 24]. In addition to single applications, benchmark suites such as OpenCL and the 13 Dwarfs[15], Rodinia [10], and SHOC [13] use GPUs as their main hardware accelerator. The main body of work, in terms of GPU applications, shows that the GPU is quite effective at generating speedups by leveraging the massive number of lightweight threads that the GPU is able to provide.

While the GPU has been able to achieve large speedups on many different types of applications, the GPU has not become a silver bullet for the entirety of the computing landscape. One of the major drawbacks found by using GPUs is that of data transfers. Datta et al. [14] show that for a stencil buffering application, a highly data-parallel graphics application, the performance difference with and without data transfers is as much as 24-fold. There are numerous other applications that claim large improvements over CPU systems only because they do not include data transfer times as part of the overall execution time, and when these costs are included, many of those applications show similar or worse performance compared to traditional CPU systems [30]. In addition, Volkov and Demmel [39] also cite data transfer performance as a large source of overhead when using GPU-accelerated systems. Harrison and Waldron [20] cite the problem of data transfer on application performance when using shaders for general-purpose computing on the GPU.

A few systems have been created to try to automatically handle communication and data transfer between the host and device for GPU systems, but none of the work is able to address the underlying problem of the PCIe bottleneck. Jablin et al. [23] present the CPU-GPU Communication Manager (CGCM), which can automatically perform data transfers for the system and optimize the performance of those transfers. The CUBA [17] and Maestro [34] frameworks also present systems which automatically perform data transfer for the user, greatly reducing the risks of error for the application. These systems were developed to improve productivity for the developer, and not necessarily to improve data transfer rates.

The new AMD Fusion architecture, described by [8] and [18], is a novel architecture released by AMD in 2011. The shared memory system and consequent hardware memory paths of the APU have been shown by Boudier and Sellers [7]. This new architecture has been shown by Daga et al. [11] to improve the data transfer performance for some applications, thereby improving overall application performance when compared to traditional GPU systems. They find that although the APU may be computationally underpowered, the amount of time saved from data transfers more than makes up for this difference in at least one application. In addition to the work of Daga et al., Spafford et al. [35] have shown the impact of AMD Fusion when compared to discrete architectures, and discuss some of the advantages and disadvantages of this system. Finally, the work of Hetherington et al. [21] use the novel APU architecture for work on the Memcached algorithm and show improved performance when using the APU system.

Many researchers have sought to model the execution times of given kernels in order to better understand optimization principles on the system. Developing an exact theoretical model to predict the performance of a kernel on a GPU is very difficult because of the massive number of threads that interact with each other as well as hide the latency of memory operations. To this point there have been two main methods for producing a model for the GPU: (i) the building of a theoretical model based on information about the hardware and runtime systems and (ii) the creation a model based on performance characteristics after running the application. As part of the former method of generating a model, Zhang and Owens [42] present a theoretical model for GPUs based on work-flow graphs to fairly accurately predict performance of GPU applications. Hong and Kim [22] present a detailed theoretical model to predict performance with a percentage error up to 5.3% of overall runtime performance. Baghsorkhi et al. [6] also present a theoretical model of the GPU system, which is able to fairly accurately predict real application performance. By using microbenchmarks to base their model, Wong et al. [40] further understand the inner workings of the GPU and use those findings to predict future performance with a predictive model. Kerr et al. [25] run a suite of 25 GPU applications and use the results of instrumentation of those runs to create a performance prediction model. None of the models presented here adequately discuss the impact of data transfer as part of the execution time. Instead the authors focus on kernel execution time alone.

In addition to previous work in modeling and data transfer performance, we investigated related work for synchronization primitives. Volkov et al. [39] presented a theoretical barrier implementation for GPU computation, but this work was never implemented on a real device. Another barrier was developed by Xiao and Feng [41] which featured both lock-based and lock-free algorithms for barriers based on the CUDA framework. Cederman and Tsigas [9] present non-blocking queuing algorithms for use in the octree application, showing the validity of the persistent threading approach and using shared memory queues.

The researchers of [38, 2] use simple spin-lock algorithms on real GPU systems. These locks show good performance and motivate some of the reasons for using global synchronization primitives on GPU systems. Stuart and Owens [37] also implement a spin lock as well as another variation of a lock and two common semaphore implementations. We use these implementations in our work to compare our non-contentious synchronization primitives against.

1.3 Contributions

The goal of this thesis is to promote the understanding of the GPU memory system and exploit that architecture to achieve better application performance. We have motivated the problem of the GPU's memory system in Section 1.1. Many of the problems present in the GPU architecture are also found throughout the heterogeneous computing field and are not just limited to GPU computing. We show that by leveraging the memory system to improve (i) data transfer and (ii) global memory contention we can achieve speedups and improve the compute efficiency of our applications. A brief description of each of these contributions is given below.

Data Transfer: We discuss the costs of data transfer as a major source of overhead for discrete GPU computing. Then we present the new AMD Fusion architecture and the trade offs on that architecture, specifically the trade off between improved data transfer performance and worsened kernel execution performance. We also investigate new techniques for data movement on the Fusion platform, each with different characteristics and performance implications. Using the AMD Fusion's improved data transfer bandwidths we are able to greatly improve the performance of data-intensive applications over discrete GPU systems due to the PCIe bottleneck. Finally, we present a theoretical model which is able to accurately predict comparative performance of different memory movement techniques for a given device. For this work we have published both a poster [29] and a conference proceedings [28] to validate our work. We have also prepared a publication submission for a journal extending our work on this topic.

Global Memory Contention: We discuss the problem of memory contention in previous and modern GPU systems. Memory contention occurs when multiple threads attempt to access the same element in global memory. For some architectures, this contention can cause accesses to be serialized, reducing the overall throughput of the system. We look at global synchronization primitives as an application where global memory contention can greatly limit performance. We show the performance impact of memory contention for different GPU and APU systems through microbenchmarking and provide ways of circumventing these performance penalties. In doing so, we present novel distributed global lock and semaphore algorithms for use in GPU computations. For this work we have prepared a conference proceeding, but have not yet submitted our work for publication.

By paying attention to the unique memory model of the GPU, we achieve speedups for applications which were previously impossible. For a data-intensive application, such as vector addition, we achieve a 2.5-fold speedup over a traditional discrete GPU by using the newer APU architecture, a computationally less powerful architecture. Similarly, we were able to achieve a more than 3-fold performance speedup using our novel distributed locking mechanism on the NVIDIA GTX 280 by avoiding global memory contention.

Fundamental Contributions: We do not know if GPUs or the new APU architecture will persist in the coming years. However, even if the AMD Fusion is rendered obsolete and is replaced by some newer and better architecture, the problem of data transfers for heterogeneous platforms will still persist as a major problem in their ability to achieve maximum performance efficiency. In addition, contentious memory accesses are, and will continue to be, a common problem for massively threaded applications. Understanding the issues present in the memory system and how the underlying architecture either aids or complicates these issues will greatly benefit application developers for these kinds of systems.

1.4 Document Overview

The rest of this thesis is organized as follows. Chapter 2 presents background information about (i) GPU computing in general, including an overview and comparison of both discrete GPU and fused APU platforms as well as an overview of the OpenCL framework used in this thesis work and (ii) a summary of the memory system problems that are addressed in this work. In Chapter 3 we present the problem of data transfers as well as a solution to greatly reduce the costs of these transfers. We also present four methods of data transfer on fusion architectures and perform a characterization of them based on four data-intensive applications. Finally, we present a theoretical performance prediction model and a comparison of fused and discrete GPU architectures for these data-intensive applications. Chapter 4 presents our work on global synchronization primitives, in which we develop novel locking and semaphore algorithms which avoid global memory contention in order to improve performance and scalability. Finally, in Chapter 5 we give a brief summary of our approaches on improving memory performance for heterogeneous architectures and discuss future work which can be performed using this thesis as a basis.

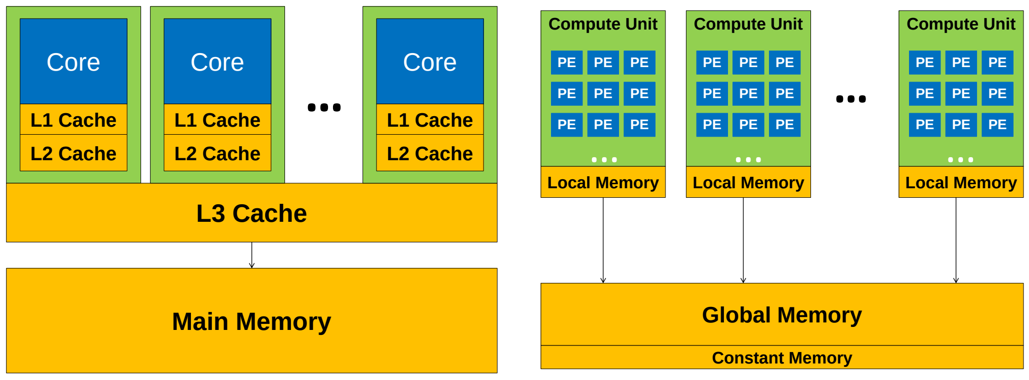
Chapter 2

Heterogeneous Computation

In this chapter we describe an overview of heterogeneous computing systems as well as the specific architectures of discrete and fused GPUs used in this work. We also discuss the problems associated with the memory system on heterogeneous platforms and their impact on application performance.

2.1 Graphics Processing Units

This section discusses the architectures of the APU and GPU. We present a detailed architectural overview of the various systems we used. We then compare and contrast the architectures of these systems and analyze the impacts on application performance. A brief overview of the OpenCL framework, used exclusively in this work, and the terminology from that framework is also described in this section.



(a) CPU Architecture (b) GPU Architecture

Figure 2.1: CPU and GPU Memory Systems

2.1.1 Discrete Graphics Processing Units

The GPU was, as its name implies, originally designed for rendering graphics efficiently to the screen. Graphics workloads are extremely data parallel and in order to facilitate faster rendering speeds more and more threads were added to the architecture to increase the throughput of this parallel workload. As graphics demands became more complex, the graphics pipeline became programmable through the use of shading languages. At this point, the originally very specific graphics hardware had become much more flexible and could be used for general-purpose computing. Because of this history, the GPU has a unique architecture when compared to traditional CPU architectures. A simplified architectural diagram is given in Figure 2.1 showing the similarities and differences of CPU and GPUs.

The CPU and the GPU were designed with very different goals in mind. The CPU's architecture is optimized to reduce latency of any given operation. As a general-purpose processor, the CPU acts as a jack of all trades in terms of performance for sequential applications. An example of this can be seen in the CPU's memory system. In order to deal with very high latency access times to memory, the CPU developed a cache hierarchy to reduce the expected time to memory access. This provides the illusion to the user of having a very fast and large memory. On the other hand, the GPU is an architecture based upon increasing the throughput of data-parallel applications. The GPU traditionally has no caching of global memory (though this has changed with some of the newer GPU architectures), and instead will aggressively perform context switching to hide the latency of these operations. The result of this architecture in the GPU allows it to achieve better throughput than a CPU system for data-parallel workloads.

A GPU is made up of multiple *compute units*, which are somewhat analogous to CPU cores, albeit far more simple than a modern CPU core. Each of these compute units contains *processing elements* that are responsible for running multiple threads in a lock-step fashion. In the GPU model, multiple threads execute the same instruction on every clock cycle. This model of computation is known as Single Instruction Multiple Thread (SIMT) computation.

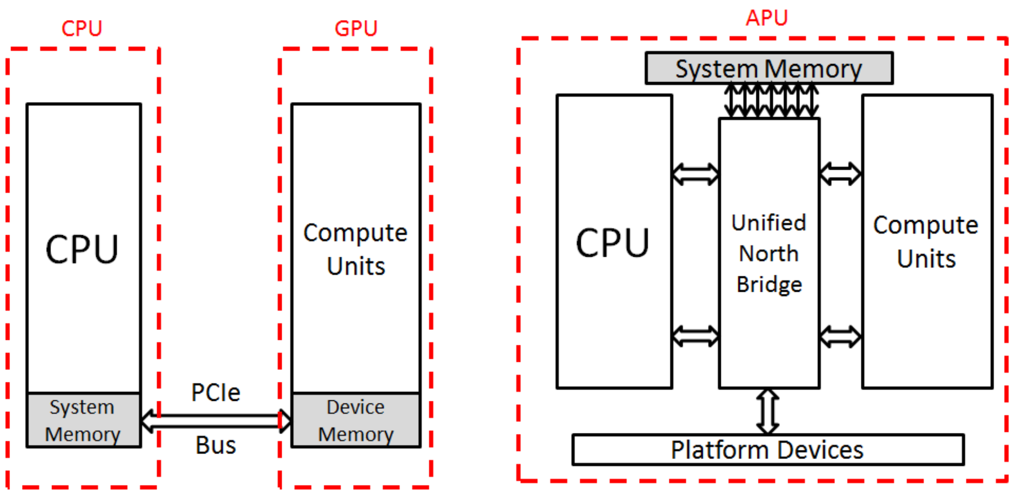
While the GPU typically does not have an intricate caching scheme like the CPU, the GPU still has a very unique memory system. The memory system is made up of four different partitions of memory, each of which has its own performance and size characteristics. The *global* memory space is shared by all of the threads on the device. That is, every thread is allowed to read or write from the global memory space. Accesses to this memory have the highest latency, typically around 400 cycles. Closely related to global memory is the *constant* memory, which is at the same level as global memory but is read-only, allowing the GPU to aggressively cache this memory. On each compute unit, there is a set of memory which can only be accessed by threads on that compute unit. This memory is called *local* memory and can be accessed with far lower latency than global memory (typically around 1-3 cycles). The GPU's local memory is used as a user-defined cache of global memory. Finally, each thread has *private* memory which consists of a register file containing the state of the thread, specifically local variables. The register file is rather large and shared on a compute unit, so the size of private memory for a thread can impact the number of threads that can be run concurrently on a compute unit.

2.1.2 Fused Accelerated Processing Units

Accelerated Processing Units (APUs), such as AMD Fusion and Project Denver from NVIDIA, offer an integrated CPU and GPU system on the same die. Because Project Denver has not yet been released, we will focus our discussion on the AMD Fusion. We anticipate that the majority of the discussion about AMD Fusion will also apply to NVIDIA's competing system. A comparison of the a typical discrete GPU system versus a fused APU system is shown in Figure 2.2.

The main focus of the APU is to integrate the CPU and GPU on the same die, thus eliminating most of the PCIe overhead when performing heterogeneous computation. Instead of accessing the device through the PCIe interconnect, the APU instead communicates between devices using a high performance memory controller. The AMD Fusion APU also has a shared memory between the CPU and GPU which is facilitated by this high performance memory controller. This memory system is still partitioned, but both devices on the APU can access either partitions of memory.

We present a detailed architecture of the APU in Figure 2.3. In addition to just adding the CPU and GPU on the same die, the AMD Fusion architecture includes the addition of two components designed to improve the coupling between CPU and GPU. The first addition to the architecture is the Write Combiner. This component is designed to improve the write speeds of the CPU to the device memory partition. The Write Combiner acts as a large buffer to reduce the number of memory transactions when writes are performed. The second major component that has been added is the Unified North Bridge (UNB). This component has two major responsibilities with respect to the AMD Fusion's memory system. Firstly,



(a) GPU System (b) APU System

Figure 2.2: Comparisons of GPU and APU systems

the UNB will perform translation from virtual GPU memory addresses to the shared physical addresses where the data is located. Secondly, the UNB is responsible for memory arbitration on the APU, ensuring that all writes get committed to the device memory properly.

2.1.3 Test Systems

Here we discuss the individual test systems that we use for this thesis work. We give a detailed discussion on the specific architectures of each of these devices and how the different architectures might impact the performance of each system. Table 2.1 gives an overview of the different discrete GPU systems and Table 2.2 shows the fused APU systems we used.

The AMD Radeon 5000-Series GPUs, also known as the Evergreen family, was released by ATI in late 2009. These GPUs feature the Terascale 2 Architecture which was produced to support DirectX 11. The AMD Radeon HD 5870 device is part of this family of GPUs. The 6000-Series from AMD, known as the Northern Island family, did not undergo any major architectural changes from the 5000-Series, but instead simply includes support for multiple

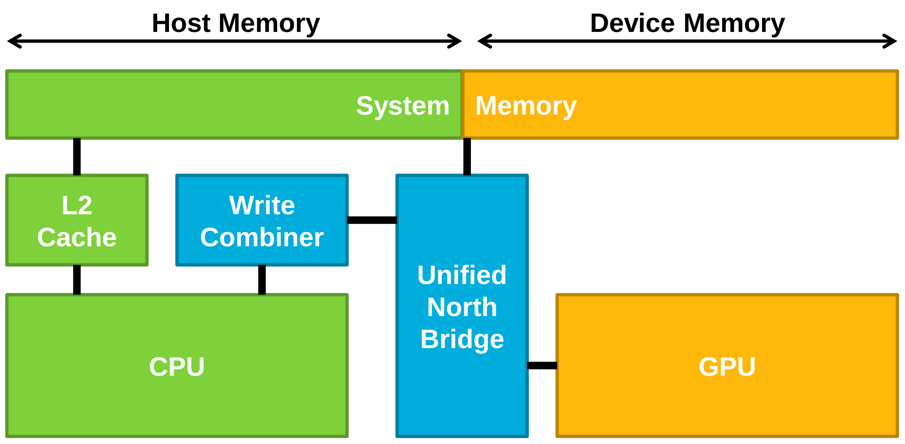


Figure 2.3: Detailed Architecture of the AMD Fusion APU

	NVIDIA GTX 280	NVIDIA Tesla 2075	AMD Radeon HD 5870	AMD Radeon HD 7970
Name	NVIDIA Low	NVIDIA High	AMD Low/Discrete	AMD High
Compute Units	30	14	20	32
Stream Processors	240	448	1600	2048
GPU Frequency	1296 Mhz	1147 Mhz	850 Mhz	925 Mhz
Max Work-group Size	512	1024	256	256
Device Memory	1 GB	6 GB	512 MB	3 GB
Local Memory	16 KB	48 KB	32 KB	32 KB
Memory Bus Type	GDDR3	GDDR5	GDDR5	GDDR5
Cache Type	None	Read/Write	None	Read/Write
Cache Line Size	0	128	0	64
Cache Size	0	229376	0	16384

Table 2.1: Discrete GPU Systems

	AMD Zacate E-350	AMD Llano A8-3850
Name	Zacate	Llano
Compute Units	2	5
Stream Processors	80	400
GPU Frequency	492 MHz	600 MHz
Max Work-group Size	256	256
Device Memory	512 MB	512 MB
Local Memory	32 KB	32 KB
Memory Bus Type	DDR3	DDR3
Cache Type	None	None
Cache Line Size	0	0
Cache Size	0	0
CPU Clock Freq	1.6 GHz	2.9 GHz

Table 2.2: Fused APU Systems

graphical outputs. The GPU cores in both fused architectures are based on this family. The Zacate machine is based on the Radeon HD 6310 GPU, and the Llano cores are based on Radeon HD 6550D system.

The AMD 7000-Series, referred to as the Southern Islands family of GPUs, represents a large departure from the previous AMD architectures. The traditional graphics-based compute units were replaced with the more general-purpose Graphics Cores Next (GCN) architecture [3]. One of the major changes brought about by this architecture is the elimination of Very Long Instruction Word (VLIW) execution. This allows more general-purpose applications to achieve greater utilization of the GPU hardware. The GCN architecture is shown in Figure 2.4. In addition to the reorganization of the individual compute units, the Southern Islands architecture also includes a coherent L2 cache for all of the global memory. This coherency allows for the device to page CPU memory and will create a tighter integration of CPUs and GPUs in future systems.

The GT 200 architecture, present on the NVIDIA GTX 280, represents the first iteration of NVIDIA's Tesla architecture. This architecture is based on Scalable Processor Arrays (SPAs). Compute units are grouped by threes into Thread Processing Clusters (TPCs). These TPCs contain a L1 cache to improve the speed of global memory reads. The NVIDIA Tesla GPUs also greatly increase the performance of atomic read, write, and exchange operations when compared to previous generation of graphics-oriented hardware.

The Fermi architecture represents the second iteration of the Tesla architecture. Present in the Tesla C2000 Series of GPUs, this compute-oriented architecture includes a configurable L1 cache, and a 768 KB L2 cache of global memory. This L1 cache can be beneficial for many compute-oriented applications where a user-defined caching scheme is impossible. In addition, Fermi also includes faster atomic operations, achieving as high as a 20-fold improvement when compared to the previous generation of Tesla architectures.

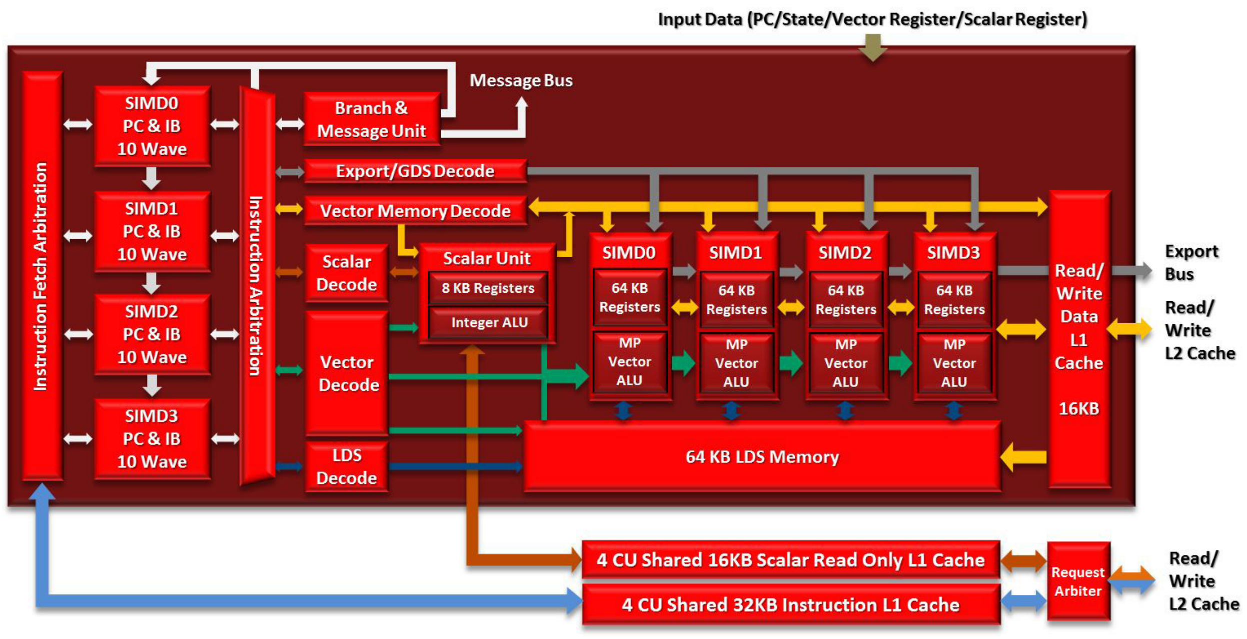


Figure 2.4: An Overview of the Graphics Cores Next (GCN) Architecture [3].

One of the major differences between the discrete GPU systems and the fused GPU systems is the type of memory used for GPGPU computation. For the discrete systems GDDR3 or GDDR5 is used, while normal DDR3 is used for APUs. GDDR3 memory is based on DDR2 memory, but includes faster read and write speeds and lower voltage requirements. GDDR5 is based on the DDR3 memory system, but also includes error checking, better performance for GPU workloads and lower power requirements. On the other hand, DDR3 memory has lower latency operations and performs more prefetching than GDDR memory, which is more helpful for single-threaded CPU workloads.

In the following sections we will investigate how these diverse architectures impact the overall performance of our applications, specifically investigating how the memory systems impact performance.

2.2 OpenCL

OpenCL is a open framework for heterogeneous computing. Developed originally by Apple, OpenCL is now an open standard under the Khronos Group [26]. This framework allows the same application code to be run on any parallel computing device which supports OpenCL, including GPUs, APUs, FPGAs, CPUs and more. OpenCL provides a familiar C-like syntax for parallel computing on the GPU, providing a major improvement in productivity over shading languages, like GLSL [27], or previous GPGPU languages, such as Brook+ [36].

OpenCL shares many traits with its major competitor, CUDA [32]. While CUDA is able to provide more advanced hardware features for its users (dynamic parallelism, GPU to GPU communication, etc.), CUDA is not an open standard and can only be used with NVIDIA GPU architectures. Because our work extends into AMD APUs and GPUs, we use OpenCL to ensure portability to those systems.

We will now present a brief overview of the OpenCL system, depicted in Figure 2.5. An OpenCL *platform* consists of multiple *devices*. Each of these devices represents an different heterogeneous device, such as a CPU, GPU, or APU. Each of these devices consist of multiple *compute units*(CUs) as well as an OpenCL *context*. The compute units are responsible for the computational power of the device. The OpenCL context is responsible for managing memory buffers on the device, as well as its work-queue.

Individual threads of execution in OpenCL are referred to as *work-items*. These work-items are then grouped into *work-groups*. Work-groups can collaborate through the use of local synchronization primitives as well as shared access to the local memory on the compute unit. In order to facilitate this collaboration between work-items, the entire work-group is pinned to the same compute unit. Multiple work-groups may be placed on the same compute unit and use the shared resources on the compute unit. However, threads from different work-

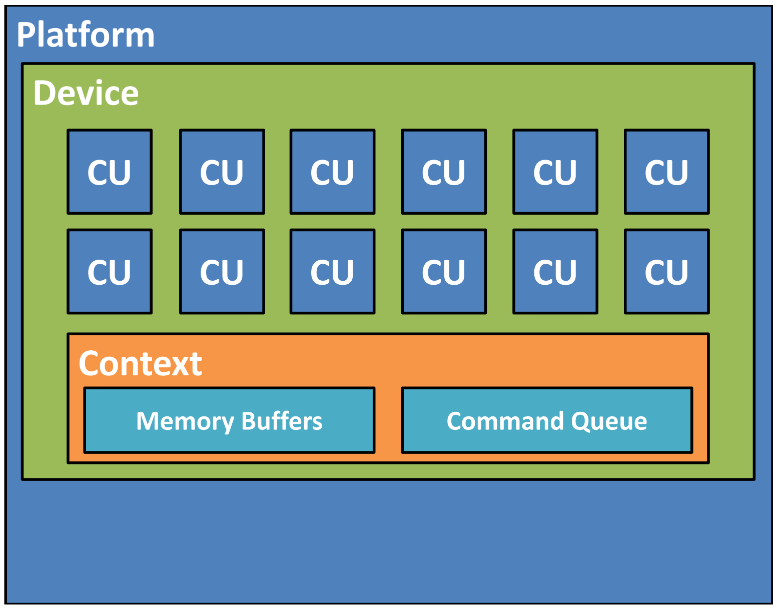


Figure 2.5: An Overview of the OpenCL Framework

groups cannot collaborate with each other to the same extent as work-items in the same work-group.

OpenCL does not guarantee any specific mapping of work-groups to compute units and this behavior should be treated as a black box. Work-groups should be able to run independently from the other work-groups running a given kernel. In many cases, however, increased communication between work-groups can be beneficial to the performance of certain applications.

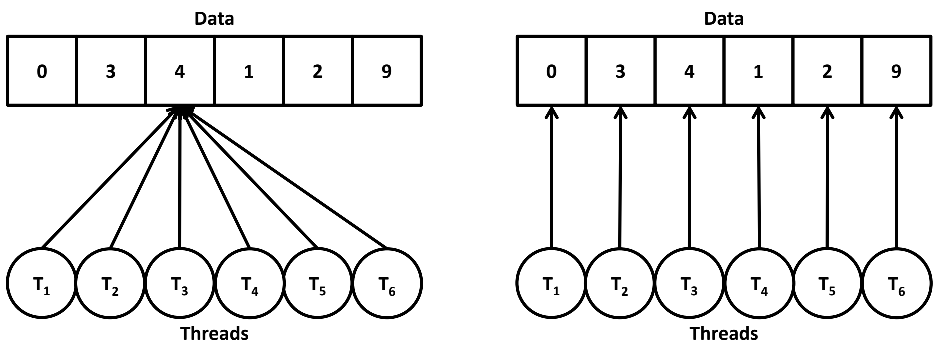
2.3 GPU Architecture Inefficiencies

GPU architectures contain a large amount of raw compute power. This is due to their SIMT-based architecture which is able to perform useful work on large numbers of threads at once. However, GPU systems are typically only able to achieve between 40% and 60% of their peak performance for general-purpose applications. Many of the architectual features that support this raw compute power actually limit performance efficiency of general-purpose applications.

We break down the problems contributing to the low performance efficiency of GPU systems into three major categories: Compute System, Programming System, and Memory System. Problems with the Compute System of the GPU mostly deal with improving the throughput of computation within a compute unit. These types of issues would include occupancy, divergent branching, and VLIW utilization. Problems in the Programming System can arise both in terms of the overhead associated with the runtime system or also the amount of programmer effort required to optimize a given application for the GPU architecture. Finally, the Memory System represents problems having to do with the movement or flow of memory in the system. Problems of this sort include data transfer and global memory contention as well as caching, coalesced memory accesses, and local memory bank conflicts. We focus our efforts for this thesis work on the Memory System, and specifically the problems of data transfer and global atomic memory contention. We present these two problems in greater detail in the following subsections.

2.3.1 Data Transfer

The PCI-Express (PCIe) interconnection is used as the path for data transfer between the CPU and GPU in discrete GPU systems. To perform computations on the GPU, the data from the host must be sent to the GPU over the PCIe bus and then the results of the computation are then returned back to the host over the PCIe bus. These two data transfers introduce a very large overhead cost for GPGPU computations, and can be so large to otherwise prohibit the application from achieving a speedup over CPU systems. The reason for this is based on the same principles as Amdahl's law [4]. The data transfers over the PCIe bus are considered part of the sequential application time. No matter how much speedup a



(a) Contentious (b) Non-Contentious

Figure 2.6: Comparisons of Contentious and Non-contentious Memory Accesses

GPU achieves on the parallel portion, the application performance will still be bounded by the sequential overhead of data transfers.

In addition to the theoretical problems of achievable speedup with data transfers to and from the device in GPGPU computing, we find that the speed of data transfer over PCIe is extremely slow. A single core sending data through the OpenCL framework is only able to achieve about 1.5 GB/s bandwidth to and from the device. For applications in which a large amount of data needs to be sent to and from the device, the costs of data transfer can become a substantial percentage of application execution time. Even for applications that still achieve speedups over CPU-based systems, the costs of data transfer to and from the device can still be extremely costly.

2.3.2 Global Memory Contention

We investigate the problem of global memory contention as a bottleneck of GPGPU application performance. Memory contention occurs on the GPU when multiple threads, specifically from separate work-groups, attempt to access the same data element in the global memory space. This can potentially result in those accesses being serialized. This serialization is present in coherent memory systems in order to ensure correct ordering of reads and writes to the memory. Figure 2.6 shows the difference between contended and non-contended global memory accesses on the GPU. In the case of atomic operations, the problem is exacerbated because by definition multiple atomic accesses to the same data must be sequentialized.

For this work we will investigate the problem of contention for global synchronization primitives on the GPU. These algorithms typically rely on busy-waits and spinning on a single value to ensure mutual exclusion. However, by having all work-groups busy-wait on a single value, a tremendous amount of memory contention occurs. This contention greatly reduces the overall speed and efficiency of those systems. A new method of synchronization primitives is needed, which can eliminate the contentious accesses without introducing significant overhead.

2.4 Contributions

In this section we outline the contributions of our work in addressing the problems laid out in the previous section.

2.4.1 Data Transfer

Using the novel fused CPU+GPU architecture, we are able to greatly reduce the amount of time required for data transfers in GPGPU applications. The elimination of the need for the PCIe interconnect allows read and write speeds to exceed those which are still bound by the PCIe bus. However, this increase in compute capability is also accompanied with less compute power than can be found on a discrete GPU system. Therefore, while the problem of the PCIe bottleneck has been solved, we must then address the trade off in terms of lost compute capability.

We show that the Fusion architecture can greatly reduce the amount of time spent performing data transfers instead of performing useful work. We perform a comparison to a traditional discrete GPU system and show that while the discrete GPU system has more computational power, the Fusion system is able to outperform it for certain data-intensive applications because of the improved data transfer speeds.

In addition to our comparison to a discrete GPU, we investigate different methods of data movement on the Fusion architecture. These data movement schemes leverage the shared memory between the CPU and the GPU. In addition to comparing some of the more intuitive movement methods, we also present a novel method for memory movement which is able to consistently perform well for our data-intensive application suite. This method exploits the fastest bandwidth paths on the architecture while avoiding bandwidth bottlenecks.

Finally, we present a theoretical model which can be used to accurately predict the best memory movement technique for a given data-intensive application and compute device. This model takes data transfer times into account and also the adjusted kernel bandwidths depending on the type of memory movement. We use this model to accurately predict performance for two of our applications.

2.4.2 Global Memory Contention

For our work, we first address the problem of global memory contention by performing microbenchmark analysis to understand the performance impact between contentious and non-contentious memory accesses on our platforms. Using this information, we produced novel distributed locking and distributed semaphore implementations for global synchronization. We show that although these new algorithms have increased overhead when compared to other approaches, the amount of time saved by eliminating the contentious accesses produces an overall application speedup for some systems.

We also investigate the use of synchronization primitives in real applications through our example application, octree. We find that the global synchronization primitives are able to outperform kernel launching techniques on at least one of our architectures.

Chapter 3

AMD Fusion Memory Optimizations

In this chapter, we investigate the specific architectural features of the APU and use those features to greatly reduce the cost of data movement on the APU system when compared to traditional GPU systems.

3.1 APU Architecture Features

As denoted in Section 2.1.2, the AMD Fusion architecture has many new architectural features which allow for a more tightly integrated CPU/GPU system. In this section we will discuss how these features can be exploited to increase the performance of data transfers. We first present an overview of how reading and writing to various memory partitions can occur on the Fusion architecture, and then discuss four ways of accessing data for GPGPU computation.

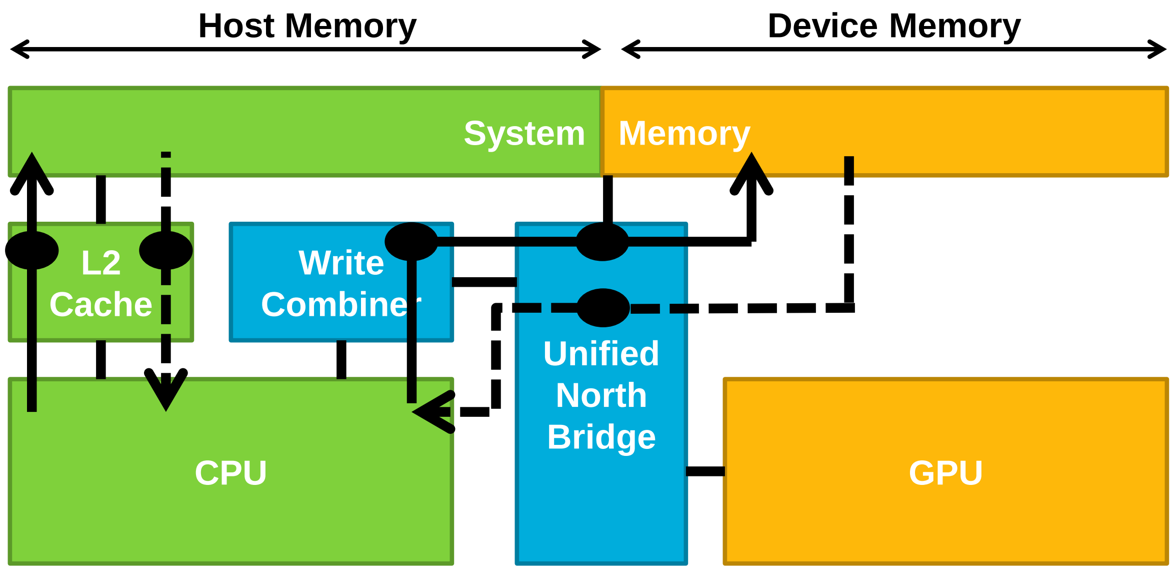


Figure 3.1: CPU Accesses. Reads are denoted by solid lines, and writes by dashed lines.

3.1.1 Memory Paths

Here we discuss the memory paths available by both the CPU and the GPU to access memory on the AMD Fusion architecture. We will also discuss the pros and cons of using different memory paths when compared to traditional memory movement techniques. The different memory access paths on the APU are depicted in Figure 3.1 and Figure 3.2.

We will first describe accesses on the APU from the CPU. These access paths are shown in Figure 3.1. Accessing host memory from the CPU is done in exactly the same way as a CPU-only system. Reads and writes to memory go through a cache hierarchy until finally committing the read or write into the system memory. Reads and writes from the CPU to device memory take different paths on the AMD Fusion. Writes to the device memory will be sent to the write combiner, which acts as a hardware buffer. When enough writes have been accumulated, one large transaction will be sent to the UNB to be finally committed into the device memory. Because of the write combiner, writes from the CPU to device

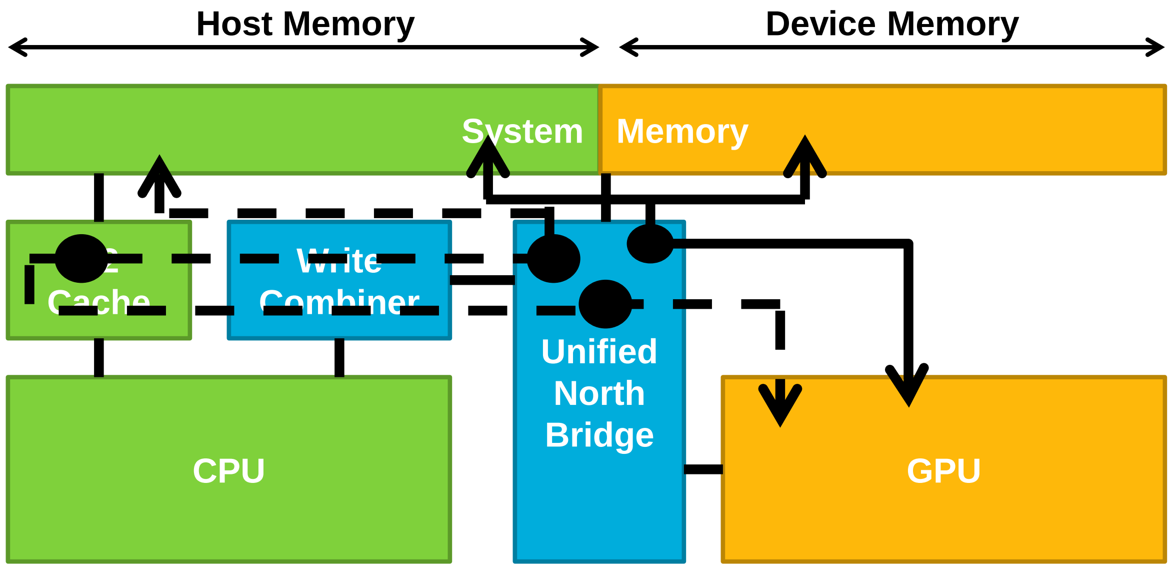


Figure 3.2: GPU Accesses. The Radeon Memory Bus (Garlic Route) is shown with a solid line, and the Fusion Compute Link (Onion Route) is shown with a dashed line.

memory have a very high bandwidth. Reads by the CPU of device memory, on the other hand, are very slow. These reads are uncached and not prefetched, which causes this path to have very low bandwidth.

On the Fusion architecture, all GPU reads and writes must occur through the UNB in order to perform address translation and to arbitrate memory accesses. The read and write paths for the GPU are shown in Figure 3.2. For accesses to device memory or to uncached host memory, reads and writes both will go straight through the UNB to the system memory. This path of memory access is referred to as the Radeon Memory Bus (Garlic Route). On the other hand, if the access is to cacheable host memory, the UNB must snoop on the caches of the CPU to ensure coherency on the CPU memory. Afterwards the access waits for arbitration in the UNB before the final commit to system memory. This path is referred to as the AMD Fusion Complete Link (Onion Route) and has a lower bandwidth when compared to the Garlic Route.

```
char* h_arr; //Initalized Host Array
cl_mem d_arr; //Already created device buffer

clEnqueueWriteBuffer(commands, d_arr, CL_TRUE, 0, size, h_arr, 0, NULL, NULL);
//Run Kernel...
clEnqueueReadBuffer(commands, d_arr, CL_TRUE, 0, size, h_arr, 0, NULL, NULL);
```

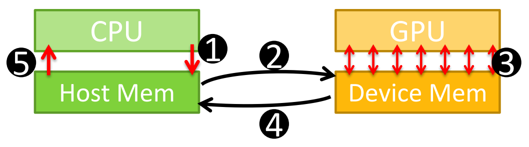
(a) Buffer Copying

```
char* h_arr; //Initalized Host Array
cl_mem d_arr; //Already created device buffer

int err;

void* d_map = clEnqueueMapBuffer(commands, d_arr, CL_TRUE,
                                   CL_MAP_WRITE, 0, size, 0, NULL, NULL, &err);
memcpy(d_map, h_arr, size);
err = clEnqueueUnmapMemObject(commands, d_arr, d_map, 0, NULL, NULL);
//Run Kernel...
d_map = clEnqueueMapBuffer(commands, d_arr, CL_TRUE,
                                   CL_MAP_READ, 0, size, 0, NULL, NULL, &err);
memcpy(h_arr, d_map, size);
err = clEnqueueUnmapMemObject(commands, d_arr, d_map, 0, NULL, NULL);
```

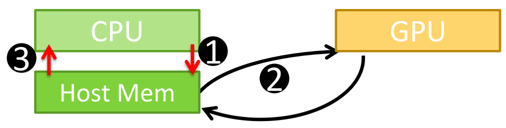
(b) Map/Unmap

Figure 3.3: Data Movement on the APU with OpenCL

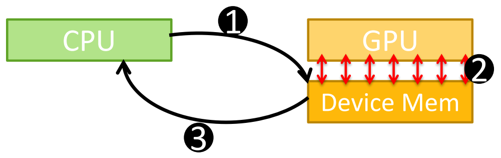
In order to use these techniques in an application, different calls to the OpenCL framework must be made. Traditionally, calles to *clEnqueueReadBuffer* and *clEnqueueWriteBuffer* would suffice. However, when performing writes directly to host memory or device memory without copying, we must use the *clEnqueueMapBuffer* interface. By mapping buffers we are able to use the zero-copy interface for AMD Fusion, in which mapping and unmapping buffers are done without performing a copy of memory. Examples of the use of both interfaces with OpenCL are given in Figure 3.3.



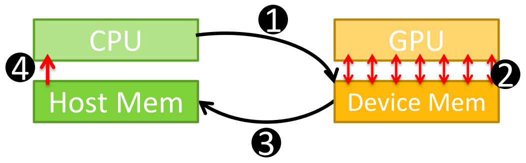
(a) Default



(b) CPU-Resident



(c) GPU-Resident



(d) Mixed

Figure 3.4: Memory Movement Technqiues

3.1.2 Memory Techniques

Based on the hardware memory paths described above, we developed four different memory techniques for the movement of data in a GPGPU application. We give an overview of these paths below and illustrate them in Figure 3.4.

The **Default** memory movement technique is the most typical technique used in GPGPU computation. This technique is depicted in Figure 3.4a. The input data for the computation begins on the host-side memory buffer. This data is copied over from the host memory to the device memory, which is then computed on by the GPU. During computation, the resultant output data set is created on the device's memory buffer. This is then copied back to the host memory. At this point, the CPU is free to read the resultant data from the host buffer. This memory access technique requires two memory copies, both to and from the device, which can be quite expensive depending on the application.

Instead of copying the data to and from the device, we can instead keep all of the memory on the host side and then let the GPU access the host memory directly. This technique is called **CPU-Resident** and is depicted by Figure 3.4b. In this case, the CPU will write the input data set to the host memory, and then the GPU will compute directly on that memory.

After the kernel computation, the resultant data will already be on the CPU-side buffer.

In contrast to the CPU-Resident case, we present the **GPU-Resident** case, in which all of the data is kept on the device buffer. This technique is shown in Figure 3.4c. In this technique, the input data set will be written directly to the device memory. The kernel will also output to the device memory and the CPU will read the results from the device memory after execution.

Because of the very slow CPU read speeds from the GPU-Resident memory case, we developed the **Mixed** memory movement technique. This technique begins in a similar way to the GPU-Resident case, where the input data is written directly to the device memory. After this occurs, the kernel will execute and produce a result on the device memory partition. Then this output data is copied over to the host memory, in the same way as the Default case, and then is read directly by the CPU. Using this technique, we never need to read data directly from the device buffer by the CPU, but instead read data from a host-side buffer. In doing so, we are able to achieve higher read bandwidth.

3.2 Methodology

In this section, we describe the experimental methodology for our characterization of memory movement on the AMD Fusion architecture.

3.2.1 Experimental Setup

For this work we used two AMD Fusion architectures (E-350 Zacate and A8-3850 Llano) and also performed a comparison to a discrete GPU architecture (AMD Radeon HD 5870). We will refer to this GPU as the "Discrete" system. An overview of the different systems that we used is given in Table 2.1 and Table 2.2. The Zacate architecture is not very powerful in terms of either CPU or GPU and represents one of the first iterations of the AMD Fusion architecture. Compared to the Discrete system, both the Llano and Zacate systems are outmatched when it comes to GPU compute power. The Discrete GPU has 4 times more compute units than the Llano system and 10 times more than Zacate. The compute units for the Discrete machine are faster than either other system. In addition, the 5870 also has a faster memory bus (GDDR5 vs DDR3). However, despite this apparent difference in compute performance, we endeavor to show that the improvements of data transfer rates for the Fusion systems will allow them to outperform the Discrete system.

All of the systems that we use for these experiments use the Windows 7 Operating system and are using OpenCL version 1.2 through the AMD APP SDK v2.6. Different CPUs can alter the system's memory bandwidth, so we will use the same CPU for both Discrete and Llano Systems. That is, we will use the CPU present in the Llano system for the Discrete system's CPU.

3.2.2 Microbenchmarks

To characterize the bandwidths of the different memory paths on the different architectures, we will use the BufferBandwidth benchmark found in the AMD APP SDK. We will measure each of the different paths as well as the default transfer speed. The BufferBandwidth benchmark will fairly accurately measure the bandwidth across different memory paths. This is done by performing multiple reads or writes in a kernel and then determining the average time per read. Having the average time per read and the size of each of those reads, we can then estimate the bandwidth over that memory path.

In addition to our bandwidth benchmark, we also analyzed the differences between the Garlic and Onion Routes in terms of effective kernel read and write bandwidth. To accomplish this, we will also run our BufferBandwidth application using these two routes to further analyze the performance impacts of the Garlic and Onion Routes. These routes are only able to be utilized for the CPU-Resident memory movement case. To use the Onion Route we will pass the `CL_MEM_READ_WRITE` flag when creating the buffer and we will pass either the `CL_MEM_READ_ONLY` or `CL_MEM_WRITE_ONLY` flags for the Garlic Route.

3.2.3 Applications

For this work we will look at five different applications, four data-intensive applications and one compute-intensive application. The four data-intensive applications are VectorAdd, Scan, Reduce, and Cyclic Redundancy Check (CRC), and the compute-intensive application is Matrix Multiplication (MatMul). We give an overview of the different characteristics of our applications in Table 3.1. In addition, we give a description of each of the applications below.

The **VectorAdd** application performs a simple vector addition $\vec{C} = \vec{A} + \vec{B}$ on two input vectors and one output vector all of length n. Each thread in our implementation is responsible for computing a single value of the output vector, performing two global memory reads, and one global memory write.

The **Scan** application computes an exclusive prefix sum vector for the vector $\vec{V}$ of length n. The prefix sum can be defined as $X_i = \sum_{k<i} V_k$, to produce the output vector $\vec{X}$ of length n. This algorithm performs two reduce-like operations to produce the result, following the computational model of [19].

The **Reduce** application will compute the sum of an input vector $\vec{V}$ of length n. This application returns only a single value which contains the sum of the vector. Each work-

Application	VectorAdd	Scan	Reduce	CRC	MatMul
Input Data Size (bytes)	$8N$	$4N$	$4N$	N	$2N^2$
Output Data Size (bytes)	$4N$	$4N$	1	N	N^2
Kernel Reads (bytes)	$8N$	$4N$	$\frac{129}{128}N$	N	N^3
Kernel Writes (bytes)	$4N$	$4N$	$\frac{1}{128}N$	$\frac{1}{256}N$	N^2

Table 3.1: Memory Characterization Benchmark Applications

group will reduce a portion of the input array based on the size of the work-group and then write the sum back to a global memory array. This application requires multiple kernel launches to completely reduce the array to a single value.

The **CRC** application is a realistic error detection algorithm which is used in networking applications to detect burst errors. Each thread in this kernel will perform one read from global memory and then will read from constant memory $log(n)$ times. Finally a work-group reduction will be performed and a single write to global memory will occur for each work-group.

Finally, the **MatMul** application will perform a matrix multiplication on two square input matrices with sides of length n. The output data is a matrix of size n^2. This application will perform n^3 reads and n^2 writes to global memory. This application is compute intensive because of the high amount of computation that occurs compared to the amount of data transfered.

3.3 Results

This section presents the results of our experiments which were described in the previous section.

3.3.1 Microbenchmarks

The results of our BufferBandwidth benchmarks are shown in Table 3.2. These results seem to validate our claims about the improved data transfer speeds of the new Fusion architectures. We see improved data transfer performance across the board for these novel architectures.

When looking at the Discrete GPU performance for our data transfer bandwidths, we notice that all of the transfers which go across partitions are bounded by the same speeds which are achievable when copying buffers to and from host and device. This is because all of those reads must actually transfer the data over the PCIe interconnect. This helps motivate the need for a fused architecture.

As we predicted, we see terrible bandwidth performance of all of the devices for reads of the device memory from the CPU. This is because these reads are uncached and not prefetched, leading to terrible performance. It is interesting to see that the performance for the Discrete system was also so bad, since the data has to be copied over from the host anyway, but this is likely done to ensure correct behavior in the drivers for all of the different architectures.

We can see the impact of the Write Combiner for the Llano and Zacate machines in the CPU write bandwidths to the device buffer. Both devices are able to write at higher bandwidths than they are able to write to host memory. This is due to the few number of memory transactions that need to occur across this path.

At first glance, it would appear that the Discrete GPU greatly outperforms the fused architectures in GPU to device buffer bandwidths. However, the reason for most of this performance difference can be related back to the larger number of compute units on the Discrete architecture. When we normalize the results based on number of cores, we find that the Discrete architecture has only double the performance of the Llano reads and a 72% increase of write performance. The remaining difference can be explained both by the different memory bus and the faster GPU core clock speeds on the Discrete system.

Transfer Type	**Zacate**	**Llano**	**Discrete**
Host Buffer → Device Buffer	1.15 GB/s	2.61 GB/s	1.25 GB/s
Host Buffer ← Device Buffer	1.18 GB/s	3.17 GB/s	1.39 GB/s
CPU ← Host Buffer (Read)	0.75 GB/s	5.67 GB/s	5.64 GB/s
CPU → Host Buffer (Write)	1.75 GB/s	5.46 GB/s	5.44 GB/s
GPU ← Host Buffer (Read)	6.49 GB/s	16.26 GB/s	1.46 GB/s
GPU → Host Buffer (Write)	3.66 GB/s	4.96 GB/s	1.28 GB/s
CPU ← Device Buffer (Read)	0.01 GB/s	0.01 GB/s	0.01 GB/s
CPU → Device Buffer (Write)	1.98 GB/s	7.49 GB/s	1.52 GB/s
GPU ← Device Buffer (Read)	6.75 GB/s	17.54 GB/s	128.74 GB/s
GPU → Device Buffer (Write)	4.78 GB/s	14.31 GB/s	98.60 GB/s

Table 3.2: BufferBandwidth Benchmark Results for Zacate, Llano, and Discrete systems. The first two rows of data represent the transfer time between the host and device memory buffers, which is over the PCIe bus for discrete GPU systems. The remaining rows represent the read and write performance of the specified processor directly on the specified memory buffer.

We also show the performance of the Onion and Garlic routes as they impact performance. The results from the BufferBandwidth application for these paths are shown in Table 3.3. The table shows that using the Onion route incurs a penalty in terms of kernel read performance, a 58% performance decrease. Because of this, we will use the Garlic route whenever possible for our applications.

Access Type	**Zacate**	**Llano**
Read (Garlic)	4.88 GB/s	16.30 GB/s
Read (Onion)	2.80 GB/s	6.81 GB/s
Write (Garlic)	2.14 GB/s	4.98 GB/s
Write (Onion)	2.16 GB/s	4.97 GB/s

Table 3.3: Garlic vs. Onion Route Performance

3.3.2 Applications

In this section we present the results of our benchmark applications using all four of the memory movement techniques. Figure 3.5 shows the results of the Llano system and Figure

3.6 shows the same results on the Zacate system.

With respect to the GPU-Resident memory, we see that for applications where there is a large or moderate amount of data to return to the host (VectorAdd and Scan), the low bandwidth of the CPU reads of device memory completely destroy the performance of those applications. However, for applications which do not send much memory back to the host, the improved CPU write speed and fast kernel speeds allow the technique to perform very well in comparison to the others. It is very important, therefore, to carefully analyze the amount of data that is to be returned to the host before using the GPU-Resident memory technique.

In terms of our performance results, we notice an interesting data point for the GPU-Resident performance data. At smaller data sizes, 20 MiB elements for VectorAdd and 30 MiB elements for Scan and Reduce, the performance of data transfers from the device to host slightly improve, and at the same time the performance of the kernel decreases slightly. The point at which the performance shifts is when the working set of the application becomes larger than 256 MiB. Before this point, we notice reduced data transfer performance but improved kernel execution. We theorize that there might be an additional partition on the device memory which gives better kernel memory bandwidth at the cost of access speed by the CPU. The Mixed data movement case also realizes this kernel performance degradation, but not the data transfer improvement. While this behavior can lead to better or worse performance depending on the application, we find that the comparative performance remains about the same because of the very low bandwidth of reads by the CPU to device memory for either of these two cases.

The CPU-Resident memory movement technique is almost the opposite of the GPU-Resident case. Across the board, kernel performance of this technique lags behind other techniques. On the other hand, the performance of data transfers are comparable or slightly better than

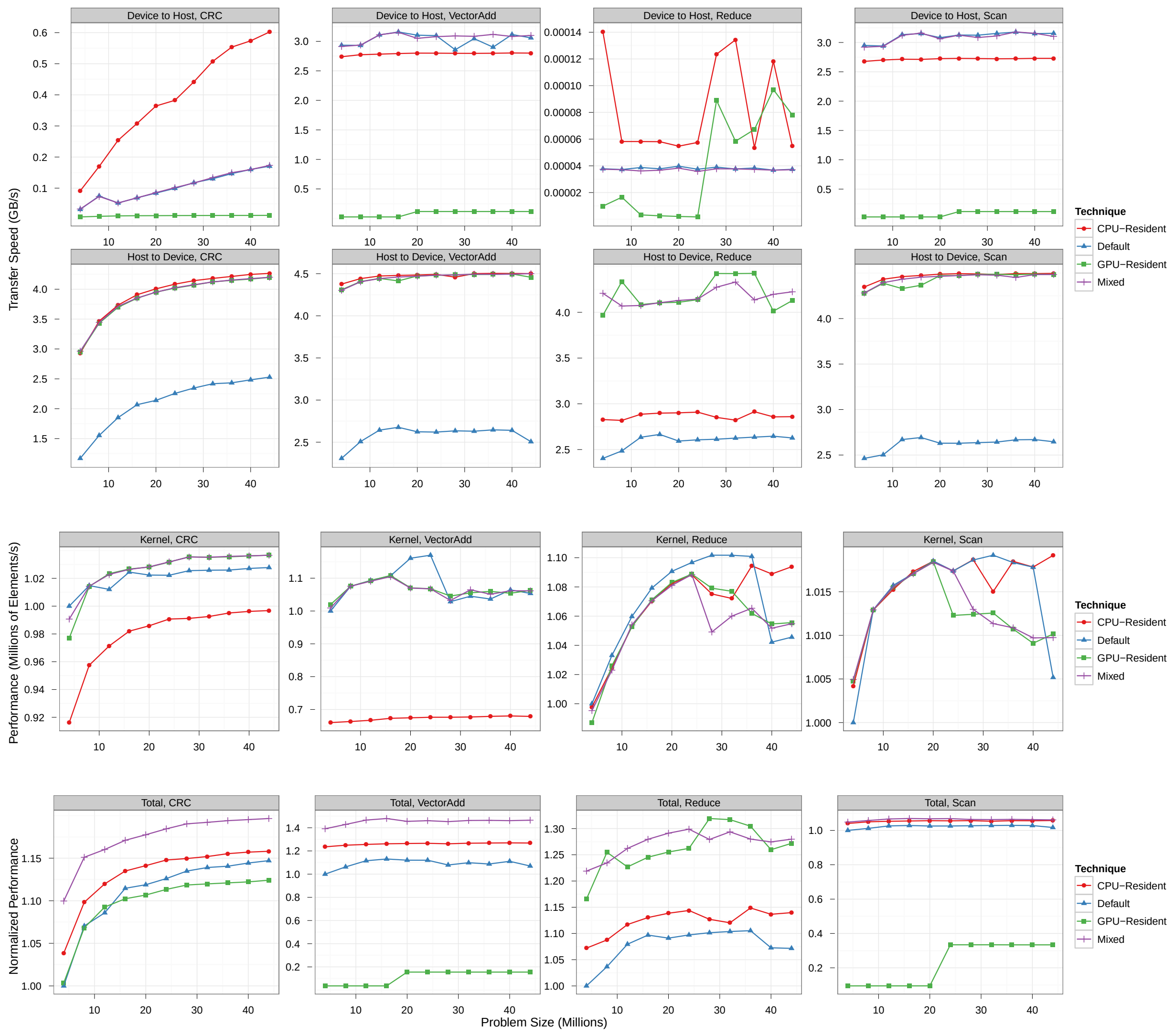


Figure 3.5: Llano Application Performance

some of the other techniques. This indicates that for applications which perform many global memory accesses, it would be best to avoid this technique.

The Mixed memory movement technique shows surprisingly good performance for all of the applications, showing either the best performance or second best performance for every application. Using the Mixed case we are able to achieve best performance in terms of CPU write speeds as well as the fastest kernel memory access speeds for the low cost of a data transfer of the result.

3.3.3 Device Comparison

Here we present a comparison between the discrete and fused architectures. We used the best performing memory movement technique for each device and application. Table 3.4 shows the techniques we used. Figure 3.7 shows the results of the comparison.

Application	**Zacate**	**Llano**	**Discrete**
VectorAdd	Mixed	Mixed	Default
Scan	Mixed	Mixed	Default
Reduce	Mixed	Mixed	Default
CRC	Default	Default	Default
MatMul	Mixed	Mixed	Default

Table 3.4: Movement Techniques used for Device Comparison

As predicted, the Zacate machine did not perform very well in our comparison with the Llano and Discrete systems. The Zacate system is very low-power and as such does not have a very powerful CPU, which impacts the data transfer rates, as well as fewer compute units when compared to the other systems. It is important to note, however, that the Zacate machine was able to outperform the Discrete system for the VectorAdd application. This is because of the large percentage of time spent performing data transfers for that application.

When comparing the Llano and Discrete systems, we see the impact of the vastly improved

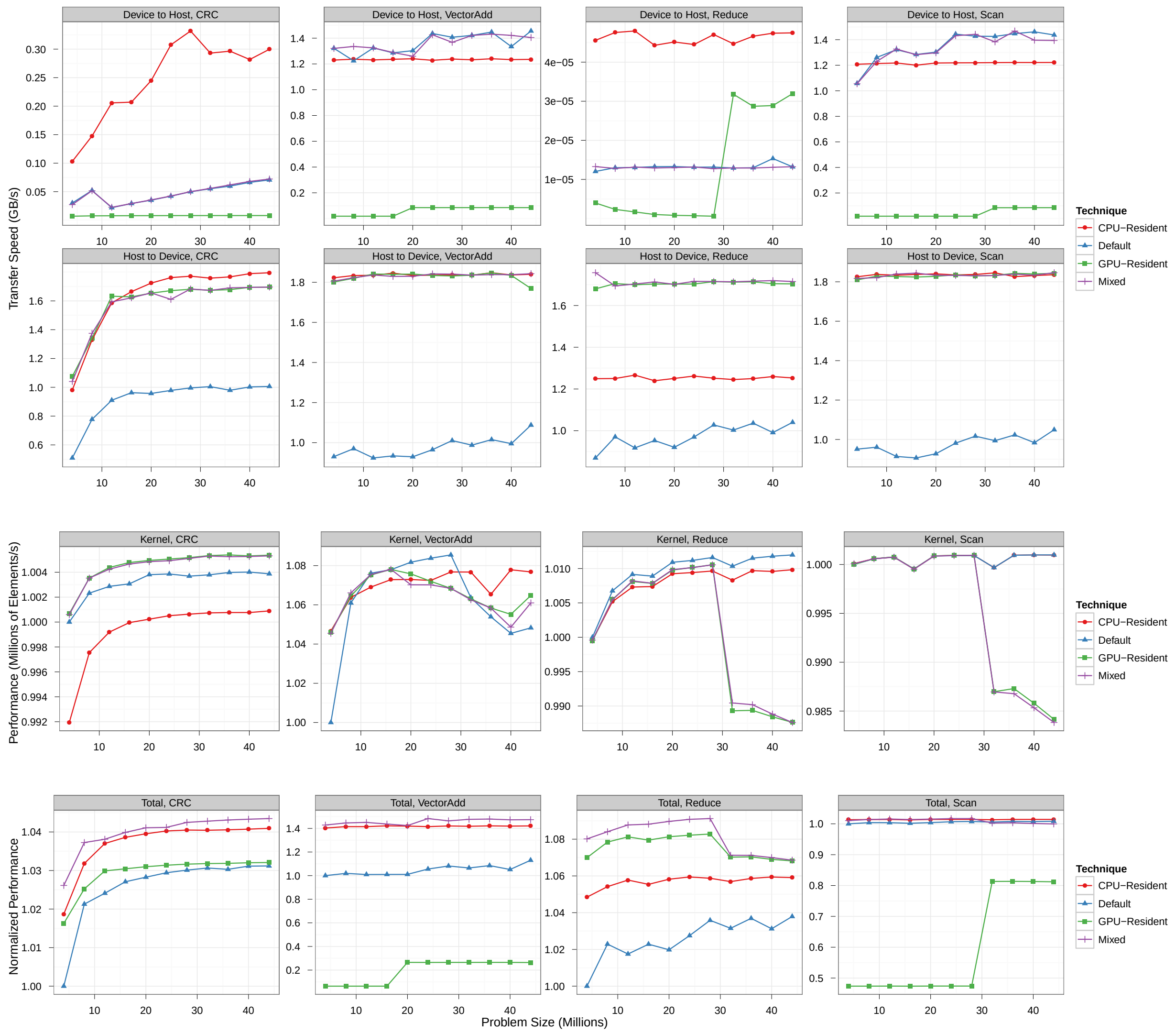


Figure 3.6: Zacate Application Performance

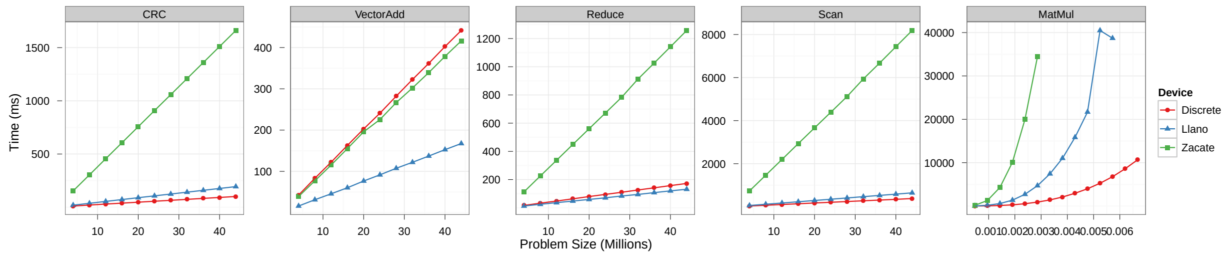


Figure 3.7: Device Comparison Results

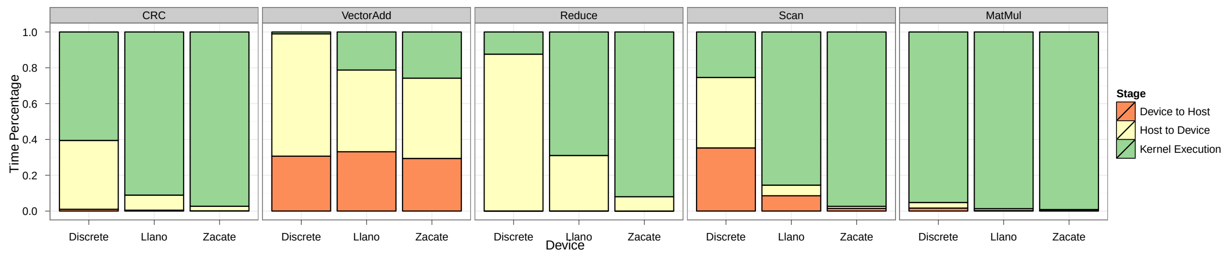


Figure 3.8: Percentage of Execution Time per Stage

data transfer rates. For the VectorAdd and Reduce applications, the Llano machine is better than the Discrete system, but for the Scan and CRC applications, the Discrete system is able to slightly outperform the fusion system. The reason for this is the greater amount of work that has to be done for those applications, allowing the faster and more numerous compute units of the Discrete system to amortize the costs of the data transfer. We would expect that if the Llano system had nearer the number of compute units as the Discrete system, that the Llano machine would again outperform the Discrete system.

The MatMul application stands in stark contrast to the other applications. For this kernel, the huge benefit of the compute power of the Discrete system is seen. Because the percentage of data transfer to computation leans heavily towards computation, the Discrete system heavily outperforms the fused architecture systems.

In addition to total performance, we looked at the percentage time spent per stage of computation for each of our applications. Figure 3.8 shows the results of that analysis. This graph shows the importance of improving data transfer for each application. For instance, the VectorAdd application spends upwards of 95% of its execution time performing data transfers. For this reason, improving data transfer speeds is of tremendous importance for that application, allowing the fusion architectures to see performance improvements even when the compared to the more powerful Discrete architecture.

We see as a counterexample the MatMul application. This application spends less than 5%

of its time performing data transfer, which while is a significant percentage, does not give the Llano or Zacate any chance of improving the overall performance of the application. This is a prime example of an application to run on a discrete GPU instead of an APU system.

3.4 Model

In this section we present a model which can be used to better understand performance of a given memory technique for a given system. For our work, we use the model to predict the best memory movement technique to use for a given application on the Llano platform. We first describe the design of the model and then validate it with real data.

3.4.1 Design

We break up the time of a given GPU application into three distinct stages: data transfer to the device, kernel execution, and data transfer from the device. We find that almost all GPU applications perform these three stages. More complex applications can, for the most part, be broken up into multiple recurrences of these stages. Using our definition, we can further define the amount of time taken for a given application with the following equation.

$$T = D_{H\rightarrow D} + K + D_{D\rightarrow H} \tag{3.1}$$

In this equation, the term T represents the total application time. $D_{H\rightarrow D}$ and $D_{D\rightarrow H}$ represent the data transfer time from the host to device and vice versa, respectively. Finally, the term K represents the amount of time required for kernel execution for this application.

We can define the amount of time taken for data transfers using Equations 3.2 and 3.3.

$$D_{H\to D} = \sum \frac{S_{H\to D}}{B_{H\to D}(S_{H\to D})} \tag{3.2}$$

$$D_{D\to H} = \sum \frac{S_{D\to H}}{B_{D\to H}(S_{D\to H})} \tag{3.3}$$

The terms $S_{H\to D}$ and $S_{D\to H}$, represent the size of the data transfer to and from the device, respectively. Similarly $B_{H\to D}$ and $B_{D\to H}$ represent the bandwidth from host to device and from device to host, respectively. We use a function to define the bandwidth of the device at a given data size. This is due to the fact the the bandwidth of a data transfer is not guaranteed to be constant, but is likely constant for very large data sizes. We investigate this claim as part of our validation.

We use a summation of individual transfers to determine the overall data transfer time. This makes the assumption that the data transfers of the application do not overlap and are performed sequentially. This assumption is usually the case for most applications, but is not guaranteed to be true in the case of asynchronous data transfers.

Finally, we look at the amount of time required for kernel execution for a given application. We break this time into three parts: global memory reads, global memory writes, and remaining computation time. Using this method we are able to derive the amount of time spent on the kernel in Equation 3.4.

$$K = \frac{S_W}{B_W} + \frac{S_R}{B_R} + C \tag{3.4}$$

In this equation, the terms S_W and S_R represent the amount of data being loaded from stored to global memory. The values of B_W and B_R represent the device's global memory bandwidth. For this model, we assume that the bandwidth of access to global memory is constant, regardless of data size. The value of C represents the remaining kernel execution time, including arithmetic operations, kernel synchronization, and local memory access. Using a simple value of C allows us to make easy comparisons between techniques on the same device and also reduces the error of these terms when performing comparisons on the same device.

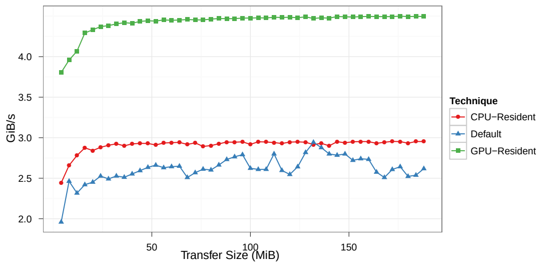
3.4.2 Validation

In addition to designing the theoretical model of different memory movement techniques on the same device, we also validated our model using real data from the Llano system. We first investigated the data transfer speeds, then used those speeds to predict data transfer performance for the VectorAdd application. We then use our model to predict the comparative performance between memory movements techniques for the VectorAdd and Reduce application.

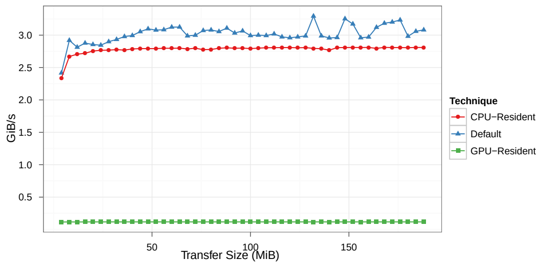
Figure 3.9 shows the data transfer rates for host to device and device to host transfers on the Llano system. The graphs show low bandwidths for smaller data sizes which improve to a nearly constant value as the amount of data transfered becomes very large. The reason for the lower effective bandwidth for smaller data sizes is the overhead associated with data transfers, while small, can have a big impact on the overall effective bandwidth of the system.

Using these figures we are able to determine a theoretical model which takes the constant overhead into account. We develop this model in Equation 3.5.

$$B(S) = \frac{S}{c + S/b} \tag{3.5}$$

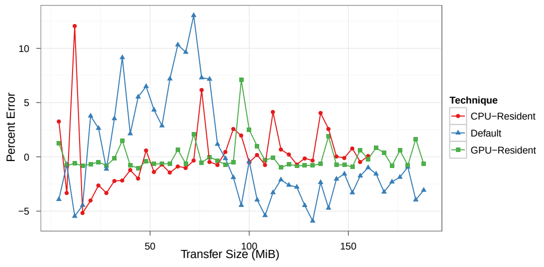


(a) Host to Device Transfer

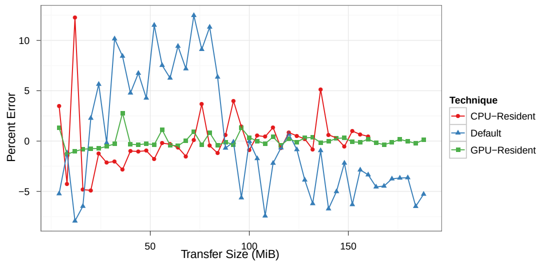


(b) Device to Host Transfer

Figure 3.9: Data Transfer Performance for the Llano System



(a) Host to Device Transfer



(b) Device to Host Transfer

Figure 3.10: Percentage Error of Llano Data Transfer Model

The value of B represents the effective bandwidth that is measured by our microbenchmark. The value of S represents the size of data being transfered, c represents the constant overhead from the transfer and b represents the true bandwidth of the transfer. We used curve fitting to determine the values of c and b for our given data sets and we plot the percentage error based on this fit in Figure 3.10. From this graph we see that most of the error for our model is within 10% and all of the error percentages fall within 15% of our predicted value. Given the variance in our effective bandwidth measurements, we consider 15% to be fairly accurate.

Using these effective bandwidth figures, we predicted the time spent an data transfer for the VectorAdd application. We determined the percentage error for each technique and each data point and plotted the result on Figure 3.11. From this graph we see that for the most part our data lands within 20% error rates, and for most sizes within 10% error. One large

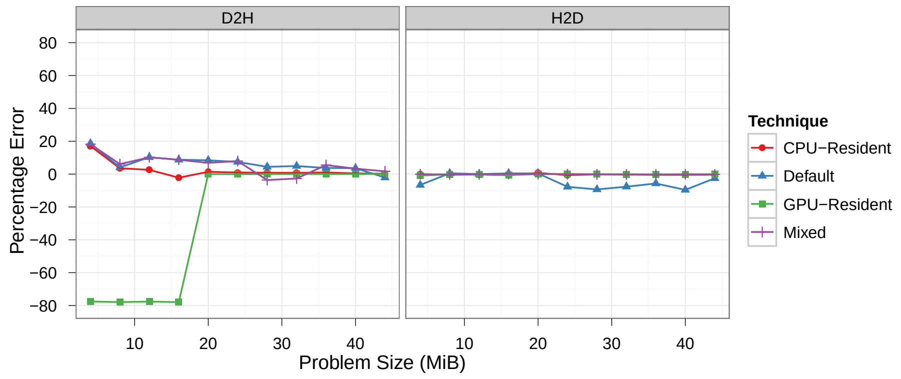


Figure 3.11: Percentage Error of Data Transfer Time for the VectorAdd Application

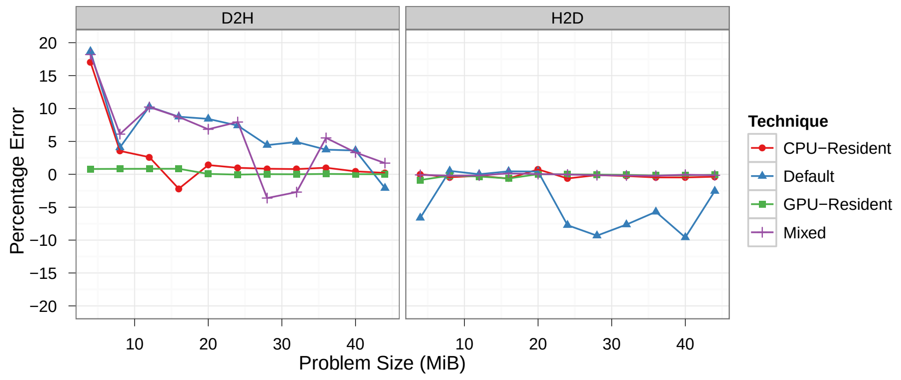


Figure 3.12: Percentage Error of Data Transfer Time for the VectorAdd Application with GPU-Resident Piecewise Bandwidth

difference we are able to see is for the GPU-Resident memory case for small data sizes. These points are around 80% off of our predictions. This is the same phenomena that we see in our original experiments comparing the techniques for the VectorAdd application. Our data transfer microbenchmark was unable to catch this phenomena as the working set for our microbenchmark was less than 256 MiB. When we take this behavior into account and use a piecewise function which will switch between 0.0256 GB/s and a linear interpolation of our microbenchmark results, we see much better error rates for the GPU-Resident case. The results of these model is shown in Figure 3.12.

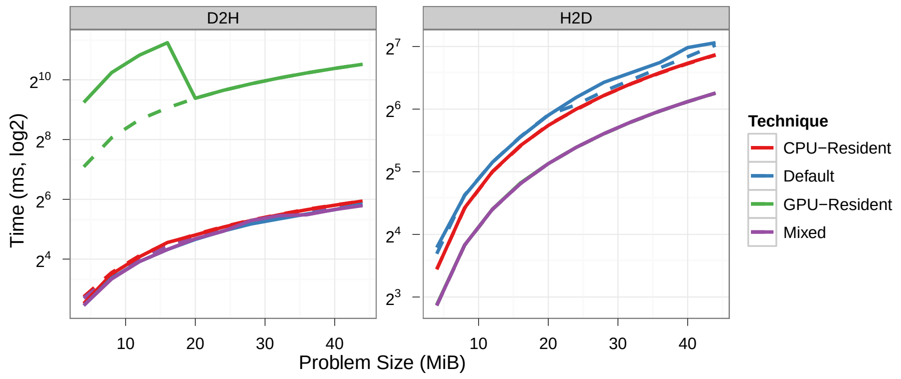


Figure 3.13: Predicted (Dashed) and Experimental (Solid) Data Transfer Times for the VectorAdd Application

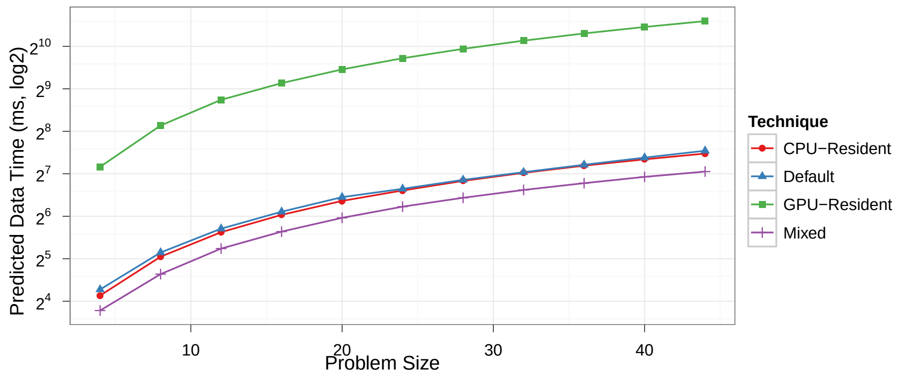


Figure 3.14: Model Predictions for the VectorAdd Application

In addition to the percentage error, we also created a graph showing the estimated and measured data transfer times for the VectorAdd application. This graph is shown in Figure 3.13. The estimated and measured data transfer times are very closely related except for the first few data points of the GPU-Resident case, as would be expected based on the percentage error.

Using the model, we attempted to predict the performance rankings of the different memory movement techniques on of the VectorAdd application on the Llano system. The results of our use of the model are shown in Figure 3.14. Our results are very promising, correctly

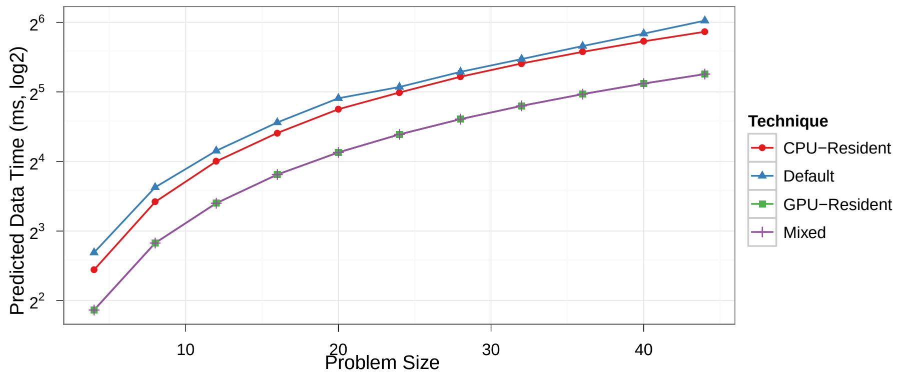


Figure 3.15: Model Predictions for the Reduce Application

predicting the Mixed memory movement technique to perform best. The model also correctly predicted the remaining memory movement techniques in their correct order: CPU-Resident, Default, and GPU-Resident. This data shows the applicability of our model for more than theoretical applications.

We also used the model to predict the performance of the Reduce application. The results of the model are shown in Figure 3.15. The GPU-Resident and Mixed cases have almost exactly the same results, which is reasonable considering how little data needs to be returned by the application. Our model predicts best performance for the Mixed and GPU-Resident memory movement cases and then the CPU-Resident case, and then finally the Default movement case. When we compare these results with our experimental results from Figure 3.5, we see that our model correctly predicted the comparative performances of our movement techniques.

Chapter 4

GPU Synchronization Primitives

In this chapter, we look at improving the performance of synchronization primitives through the reduction of contentious memory accesses. In doing so, we produce a novel locking mechanism for global synchronization primitives on GPUs.

It is important to note that all of the synchronization primitives we discuss here work at the work-group level of granularity. That is, we assume that there is only one active thread on the work-group when a call to the synchronization primitive is made. If this is not the case then deadlocks can occur based on how the hardware schedules threads running in a work-group. This limitation could cause problems for applications where very fine-grained parallelism is required, but should be sufficient for most of the problems necessitating synchronization on the GPU.

4.1 Traditional Synchronization Primitives

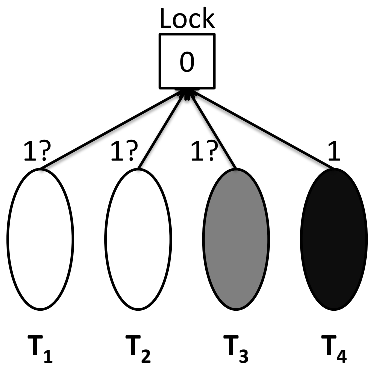
In this section we look at initial attempts of global synchronization primitives for use on GPUs. Specifically, we are looking at the lock and semaphore implementations of Stuart and Owens [37]. Though these primitives were not the first or only implementations, they represent some of the most common approaches for synchronization primitives used in GPGPU computing.

4.1.1 Locks

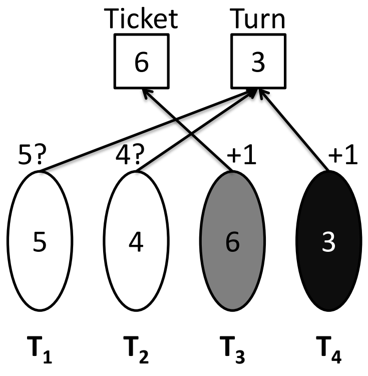
There are two different locking mechanisms that we will investigate for this work: the **Spin Lock** and the **Fetch-and-Add (FA) Lock**. Both of these algorithms depend on contentious memory accesses on a single variable.

The **Spin Lock** is the simplest lock in terms of both code and data footprint. Every thread wishing to lock will simply atomically exchange with the locking variable until it returns an unlocked value. To unlock after computation, the thread must simply atomically exchange back the lock variable to the unlocked state. A picture of this lock is shown in Figure 4.1a. One of the downsides of this lock, however, is that starvation can occur. It is possible that a thread trying to lock will continuously get preempted by other threads and will never be able to acquire the lock until all of the other threads have completed execution. This could cause improper load balancing for an application.

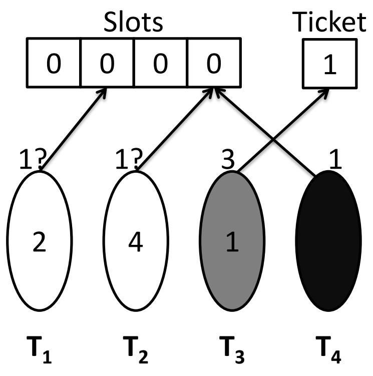
The other locking mechanism that we investigate is the **FA Lock**. This lock is similar to the Spin Lock, but solves the problem of starvation occurring, and is shown in Figure 4.1b. To lock, each thread will atomically increment a ticket variable, giving the thread a unique ticket. The thread will then continuously atomically exchange with a turn variable until it



(a) Spin Lock



(b) Fetch-and-Add Lock



(c) Distributed Lock

Figure 4.1: A depiction of the lock implementations used for this work. In each subfigure, thread T_4 has acquired the lock and is in the process of unlocking. T_3 has just begun to locking procedure, illustrating the startup procedure of that lock.

is equal to the thread's designated ticket value. When this occurs, the thread has obtained the lock. To unlock, the thread must simply atomically increase the turn value of the lock. This lock only requires two global integer variables to work. This lock also guarantees that starvation will not occur as the maximum time for a thread to wait is equal to the number of work-groups using the locking mechanism.

4.1.2 Semaphores

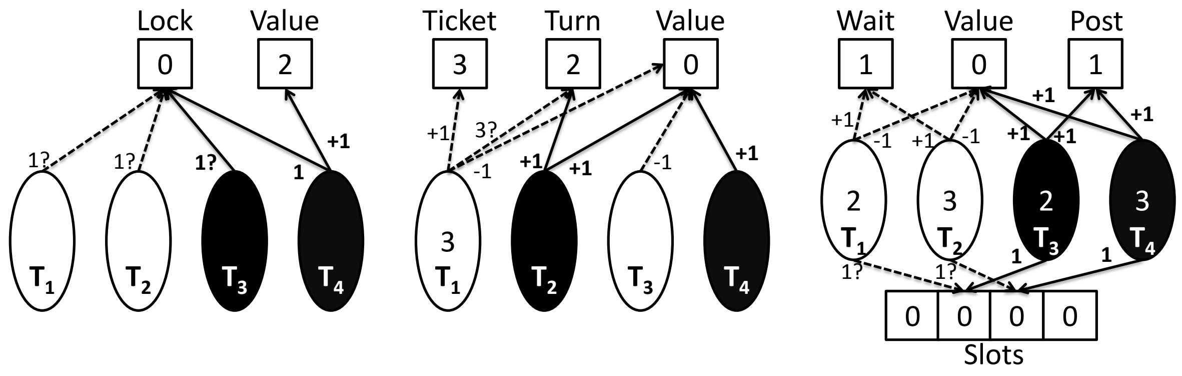
We use two semaphore implementations to compare our new distributed semaphore to. These semaphores are referred to as the **Spin Semaphore** and the **Sleep Semaphore**. As with the locks in the previous subsection, both of these semaphores will perform spinning on contented global variables. We describe these two semaphore schemes below.

The **Spin Semaphore**, shown in Figure 4.2a, is fairly straightforward in terms of semaphore implementations, but this implementation can be prone to deadlock on some systems. Both posting and waiting for this semaphore require acquiring a lock and spinning on the locking mechanism. To wait using this semaphore, a thread will constantly acquire the lock and then test to see if a value variable is greater than 0. If it is, the thread will decrement this variable, unlock and then continue execution. Otherwise, the thread will unlock and continue to spin. On a post, the thread must acquire the lock and then increment the value variable. In this system there is no guarantee that the semaphore will complete execution if there is at least one other thread waiting, because of the starvation problem that can occur. In addition, this scheme incurs a lot of overhead from the interference between posting threads and waiting threads. For these reasons, we believe that the Spin Semaphore is typically not the best solution for semaphore synchronization on the GPU.

The other semaphore implementation that we will investigate is the **Sleep Semaphore**, depicted in Figure 4.2b. For this semaphore, each waiting thread will receive a ticket by atomically incrementing a ticket variable. Then they will wait for the turn variable to be equal or greater than their ticket, in much the same manner as the FA Lock. Afterwards, they may continue execution. To post, the turn thread must simply be incremented. This semaphore avoids the problems of the Spin Semaphore presented above, but can have more overhead in the waiting phase. The waiting phase still involves a contentious spin loop on global memory. In the posting phase, the amount of overhead is substantially less when compared to the Spin Lock, requiring only a single atomic increment instruction.

4.2 Distributed Primitives

In this section we present the design of our distributed locking and semaphore algorithms. These algorithms were designed to avoid the atomic memory contention problems of the above primitives, which causes both increased performance as well as scalability.



(a) Spin Semaphore (b) Sleep Semaphore (c) Distributed Semahpore

Figure 4.2: The Semaphore implementations used for this work. Threads filled in are posting to the semaphore while the other threads are waiting on the semaphore.

4.2.1 Distributed Lock

The Distributed Lock (D-Lock) algorithm contains a novel distributed network to avoid contentious atomic memory access while also ensuring that starvation does not occur. This locking scheme is depicted in Figure 4.1c and the algorithm for this lock is shown in Algorithm 1.

To lock, a thread, T, will atomically exchange their group ID with a variable in the mutex. The returned value will be the group ID of the last thread to begin acquiring the lock. T will then wait for the previously acquiring thread to unlock the lock. At this point, T has acquired the lock and can continue computation. We make use of an array to avoid contentious atomic accesses for this lock. Each work-group will have a specific slot in the array based on its group ID. The acquiring thread, T, will constantly check the slot of the preceding thread to see if it has unlocked, and when it does, it will reset that slot of the preceding thread and then continue execution. To unlock, the thread must simply set the state of its slot to unlocked to allow the next thread to continue.

In terms of space required, the D-Lock algorithm presented here will require one integer to act as the turn variable and an array of size equal to the number of work-groups launched. This is more than the naive lock algorithms require, but is still a small amount when compared to the amount of global memory on the system. Global synchronization primitives are typically utilized in the persistent threading paradigm, so the number of slots needed in D-Lock is likely to be small.

In terms of performance overhead, this method is roughly equivalent to the FA Lock and must do one extra atomic exchange than the Spin Lock. However, this slight overhead increase also completely eliminates the remaining contentious atomic memory reads. This should make the algorithm much more scalable than either of the other two algorithms.

Algorithm 1 Distributed Lock Algorithm

```
Let $a\_xchg$ represent an atomic exchange
function LOCK(mutex $m$)
    $bid \leftarrow group\_id()$
    $watch \leftarrow a\_xchg(m.ticket, bid)$
    while $a\_xchg(m.slots[watch], 1) \neq 0$ do
    end while
end function

function UNLOCK(mutex $m$)
    $a\_xchg(m.slots[watch], 0)$
end function
```

4.2.2 Distributed Semaphore

We present a novel distributed semaphore called D-Sem. Like the D-Lock algorithm described above, this algorithm also makes improvements on existing semaphore algorithms by eliminating the atomic global memory contention problem. However, this is done at the cost of extra overhead both in terms of computation as well as memory footprint.

In order for a thread to wait on this distributed semaphore, it will first check to see if it is allowed to continue execution by checking a value variable. If this value is greater than 0 before an atomic decrement takes place, then the thread is allowed to continue execution. Otherwise, the thread will get a ticket and wait on a corresponding slot in a shared global array. This wait is done with a spin loop using a atomic exchange operation. When this slot is changed from the locked to unlocked state, the thread is allowed to continue its execution.

Posting on the thread is somewhat similar to waiting. The posting thread will increment a global value variable and if that value was less zero before the decrement, a ticket indicating the next thread to unlock will be incremented and the array slot associated with that ticket will be unlocked.

Similar to the D-Lock implementation, our implementation of D-Sem requires an array of values in order to eliminate memory contention. However, this also adds a larger memory footprint for the algorithm. Luckily, this overhead as the number of slots can be any length longer than the number of threads which can run concurrently on the device, or more conservatively, a number equal to the number of work-groups. The conservative use of an array size equal to the number of work-groups is sufficiently small in a persistent threading model, but could be problematic if such a model is not used.

In terms of algorithmic overhead, our D-Sem implementation has more overhead than the Sleep Semaphore. In the case of posting, extra steps need to occur to determine and then set the specified slot value, while in the Sleep Semaphore, only a simple increment of a variable needs to take place. However, depending on the number of threads waiting on a semaphore, we believe that our solution may yield better results by eliminating the atomic global memory contention.

Algorithm 2 Distributed Semaphore Algorithm

```
Let $a\_xchg$ represent an atomic exchange
Let $a\_inc$ represent an atomic increment
Let $a\_dec$ represent an atomic decrement
function WAIT(semaphore $s$)
    $v \leftarrow a\_dec(s.value)$
    if $v < 0$ then
        $watch \leftarrow a\_inc(s.ticket)\%s.slots.length$
        while $a\_xchg(s.slots[watch], 0)$ do
        end while
    end if
end function

function POST(semaphore $s$)
    $v \leftarrow a\_inc(s.value)$
    if $v <= 0$ then
        $free \leftarrow a\_inc(s.turn)\%s.slots.length$
        $a\_xchg(s.slots[free], 1)$
    end if
end function
```

4.3 Methodology

In this section we describe the methodology of our experiments which will attempt to prove the validity of both our claim of atomic global memory contention being a problem as well as using that knowledge to impact the performance of global memory synchronization primitives. Finally, we present a realistic octree creation application and wish to show the validity of synchronization primitives on a real application.

4.3.1 Experimental Setup

For this work we will use both AMD and NVIDIA GPUs so that we can find the true extent to which our new algorithms improve upon existing ones. We use the two AMD and two NVIDIA GPUs described in Table 2.1. For each vendor, we use both a commodity graphics

GPU, referred to as the Low machine, as well as a more compute-oriented GPU system, referred to as the High machine. In addition to these platforms we also include the AMD Fusion architecture, Llano. We anticipate that the results of the Llano machine will be similar to that of the AMD Low machine because the fundamental architecture is the same between the machines, but our results might differ because of the different memory used in the fused architecture, DDR3.

All of the work for this experiment was done using OpenCL 1.1 because it can be used across both AMD and NVIDIA platforms. All of the tests were performed using the Linux Operating System using the latest drivers available for each system, CUDA 4.0 for the NVIDIA machines and AMD APP SDK 2.7 for the AMD systems.

4.3.2 Microbenchmarks

Here, we describe the microbenchmarks we will use in order to test the validity of our claim of atomic memory accesses being a problem on GPU systems. We will also look into different atomic instruction performance on each system as well as varying stride length to further reduce memory contention. Finally, we will use microbenchmarking to determine if our new D-Lock and D-Sem implementations can outperform the naive global synchronization primitives.

First we will use microbenchmarks to determine the difference in performance between contentious and non-contentious reads on the GPU. We will launch enough threads to fully occupy the device and then perform 1000 atomic operations. For the contentious accesses, all of the accesses will be to a single value. For non-contentious access, each thread will perform accesses on its own global data. Based on the information of these microbenchmarks, we are able to determine the approximate bandwidth of these operations on the device.

Atomic reads are performed using the `atomic_add` instruction with a value of 0, and atomic writes are performed in a similar way using `atomic_xchg` where the value returned by the exchange is never used.

In addition to reading and writing, we also looked at the comparable performance of different atomic operations on each device. Specifically, we looked at the following atomic instructions: `atomic_add`, `atomic_inc`, and `atomic_xchg`. We wanted to determine if atomic performance was consistent between instructions or if some instructions had more overhead than others.

We also wanted to investigate to what degree global contention was happening for accesses to neighboring data. The performance of accessing data elements near to other access could be lower due to caching effects on the GPU. We therefore tested different stride lengths for accesses to global memory, the distance between neighboring thread accesses. We tested different stride lengths from one to 1024 by powers of two to understand this phenomenon.

Finally, we implemented the three lock and semaphore implementations previously described and ran them multiple times in order to understand how their performance scales with the number of work-groups being run. Each work-group launched performed either a lock and unlock or a wait and post 1000 times before ceasing execution. Each kernel was run 1000 times and the mean of all the runs was determined.

4.3.3 Application

In order to test the applicability of synchronization primitives on realistic applications, we use the octree algorithm as an example application. This algorithm was used by Cederman and Tsigas[9] in their work on dynamic GPU queuing. The octree algorithm is a spatial partitioning algorithm for a given set of points, which is used extensively in graphics applications and physics simulations. It is important to note that this algorithm uses a shared queue as part of its implementation.

At the beginning of the computation, all of the points in the input set are considered. These points are then sorted based on their spatial octant. Each octant created will continue to be split if the number of points in that octant is greater than a specified threshold. Once this threshold is met for all octants, computation is completed.

We compared our synchronization primitives against a naive Kernel Launch method, in which all synchronization occurs implicitly through kernel execution barriers. This method also communicates with the CPU between executions to determine if computation has finished.

Our lock-based approach does not require any global barrier. Instead accesses to a work-queue containing an octant that needs to be partitioned will be shared by all the threads. Threads will pull work off of this queue using locks to ensure no simultaneous access is occurring. When threads produce additional work, they will enqueue it on to the shared work-queue. Execution ends when there are no more items left on the work queue and every thread is waiting on that queue for additional work.

4.4 Results

In this section, we present the results of both the microbenchmark data as well as our realistic octree application data.

4.4.1 Microbenchmarks

The results for our microbenchmarks on global atomic accesses are given in Figure 4.3. These results show that contentious accesses incur a large performance penalty when compared with their non-contentious counterparts. For the AMD Low and AMD High machines, we notice

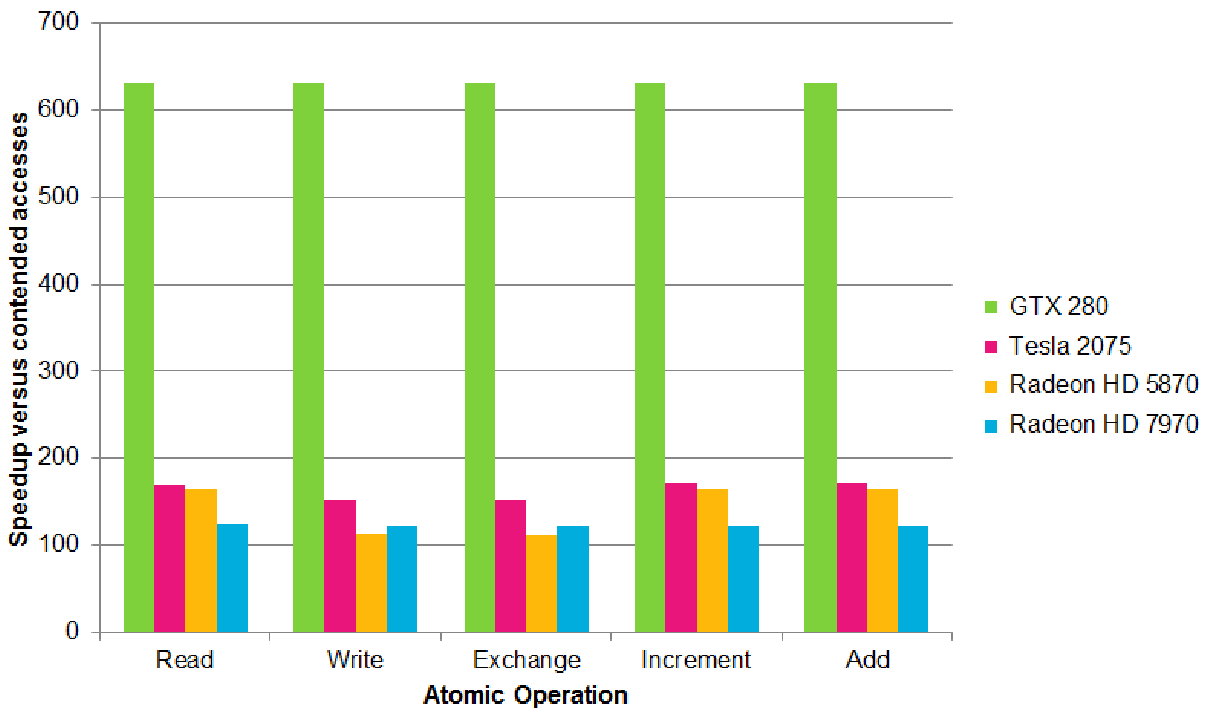
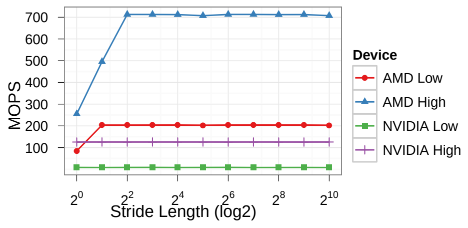
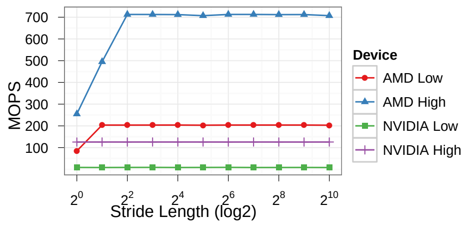


Figure 4.3: Atomic Instruction Comparison between Contentious and Non-Contentious Accesses



(a) Read



(b) Write

Figure 4.4: Atomic Performance with Varying Stride

a 165-fold and 125-fold performance difference, respectively. On the NVIDIA architectures we see a much more staggering difference of 170-fold and 630-fold performance difference for the High and Low machines, respectively. The results of this microbenchmark show the importance of eliminating contentious atomic memory accesses as much as possible.

Another surprising figure from these results are that of the Atomic Exchange performance on the AMD Low system. We notice almost an 1.5-fold slowdown in atomic performance when using the atomic exchange operation. This information could be important for the design of future algorithms using atomics on that system. The other systems that we tested show only negligible differences between various atomic operations.

Figure 4.4 shows the performance when using different stride lengths for atomic reads and writes. We notice that in terms of writing performance, none of the devices are dependent on stride length for increased performance. On the other hand, we notice an increase in atomic read performance as the stride length increases for the AMD systems. After increasing the stride length to 4 for the AMD High machine and 2 for the AMD Low machine, we see constant performance for the atomic read operations. For this reason, we ensured the use of strides greater than 4 for our D-Lock and D-Sem implementations on the AMD machines.

The performance of our lock implementations for the four systems we tested are shown in Figure 4.5. The results show a surprisingly good performance for the Spin Lock implemen-

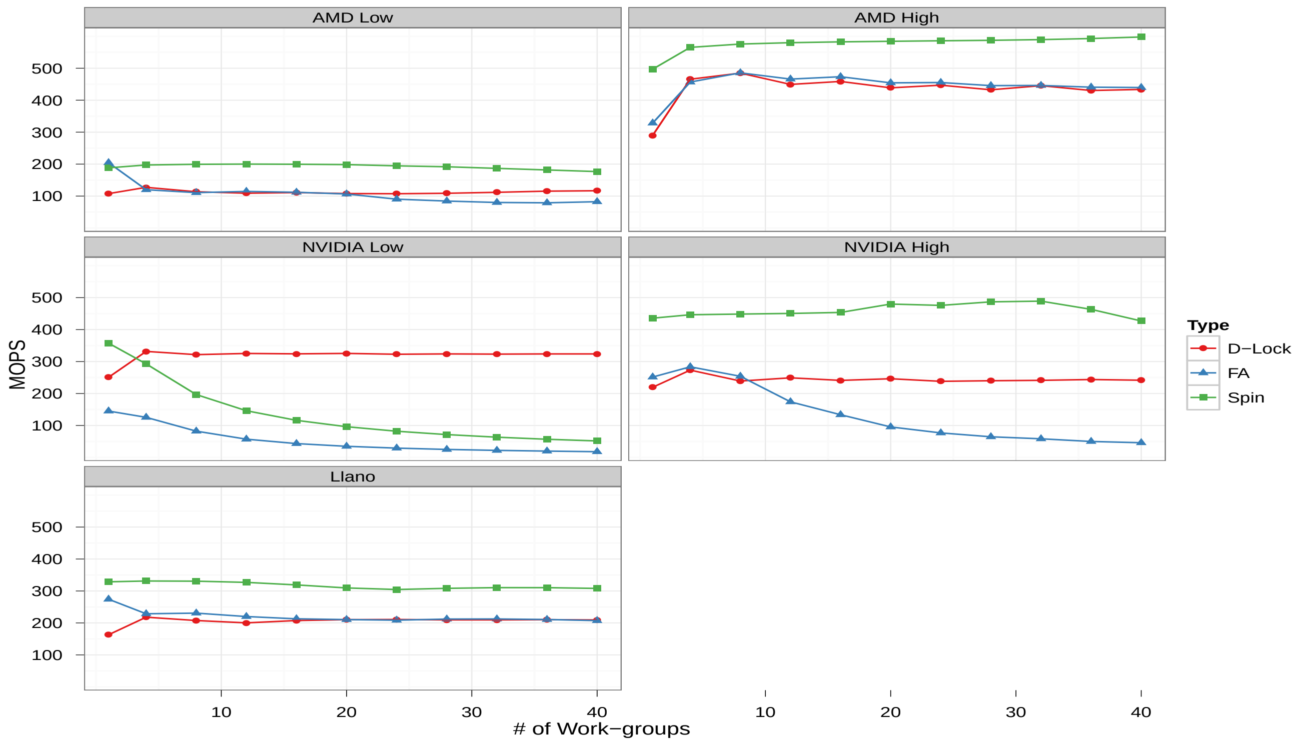


Figure 4.5: Lock Performance for Varying Work-group Sizes

tation. The reason for this good performance is most likely due to the reduced overhead of that implementation. One the NVIDIA Low machine, we see a drastic loss in performance of the Spin Lock as the number of work-groups increases, this is caused by the contentious memory accesses performing so poorly on that system (630-fold performance difference).

When comparing the performance of D-Lock against the FA lock, our D-Lock implementation generally outperforms the FA lock as the number of work-groups increases. In general, we see that the performance of D-Lock is consistent across number of work-groups whereas the FA Lock decreases in performance. This shows the scalability of our distributed lock implementation when compared to other locking schemes.

We show the results of our semaphore microbenchmarks in Figure 4.6. For these tests, we notice that the performance of the Spin Semaphore is very poor on both Low machines. We were also unable to run the Spin Semaphore on the AMD High machine without crashing. On both low machines the amount of overhead caused by the spinning while locking and unlocking a mutex is far too large to produce any kind of performance benefit. This caused performance to plummet on all of the systems besides the NVIDIA High system. We believe that the NVIDIA High machine was able to achieve good performance on the Spin Semaphore because of the vastly increased atomic performance on NVIDIA Fermi architectures. This improved performance could reduce the amount of contention for the lock by simply having a vastly superior throughput, ensuring that few threads are attempting to acquire the lock at a given time.

Our D-Sem algorithm performs best on the NVIDIA Low machine because of the effects of contentious atomics on that system. On the other hand, for AMD systems, the time saved by D-Sem in removing contentious memory accesses was not enough when compared to the overhead required for our new algorithm. For this reason, the standard Sleep Semaphore performs better on those systems.

Comparing the Llano and AMD Low performance for both locking and semaphore algorithms, we notice the same trends for both systems, and comparatively similar performance. We believe that the larger number of compute units in the AMD Low machine actually worked against it for these tests by increasing the amount of contention. The fewer compute units of the AMD Llano machine reduce contention, causing increased performance for that system.

4.4.2 Octree Performance

We compared the performance of the Kernel Launch and lock-based methods on two data sets for the octree application. The results of these tests for a uniform data set are shown

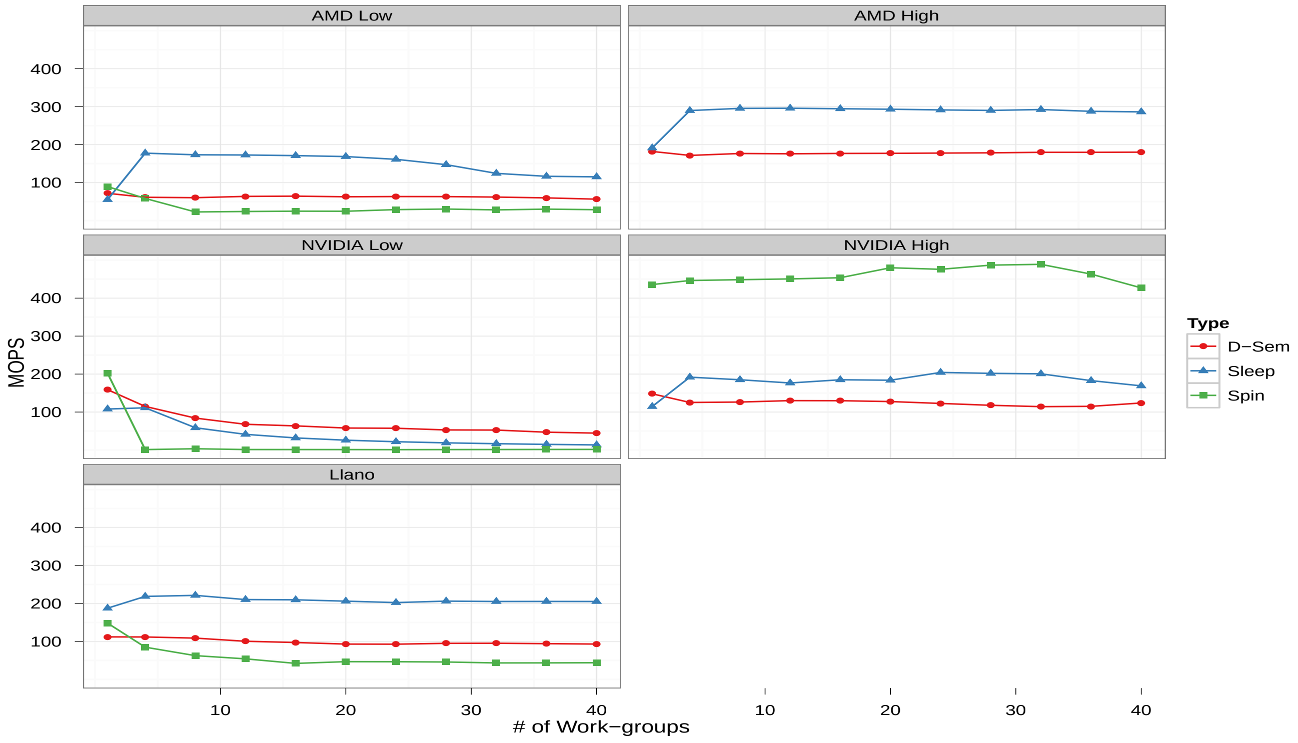


Figure 4.6: Semaphore Performance for Varying Work-group Sizes

Device	Kernel Launch (ms)
AMD Low	0.161
AMD High	0.157
NVIDIA Low	0.087
NVIDIA High	0.124

Table 4.1: Kernel Launch Times

in Figure 4.7 and for a cylindrical data set in Figure 4.8. For these results we used the Spin Lock and the Sleep Semaphore for all systems.

In addition, we also ran a microbenchmark to determine the amount of time required to launch a kernel on each of these systems. The results of that microbenchmark are shown in Table 4.1. The kernels launched were empty and were run 100 times before an average result was taken.

The results for both of the data sets show the same trends in terms of overall performance. Only the AMD Low machine vastly outperforms the Kernel Launch method with the lock-based one. Both the Spin Lock and Sleep Semaphore performed the best for this system. In addition, the time spent to launch a kernel was highest for this device. This lead to the lock-based implementation outperforming the Kernel Launch version.

On the other hand, when looking at the performance of the NVIDIA Low machine we see that the performance of the lock-based method was far worse than the Kernel Launch method. The reason for this discrepancy is the exact opposite of the AMD Low machine. The kernel launch times for the NVIDIA Low machine were the lowest of any of the systems we tested by a wide margin. In addition, the Spin Lock performed poorly on that system. Using our D-Lock and D-Sem algorithms we would expect to see improved performance, more closely matching the Kernel Launch performance.

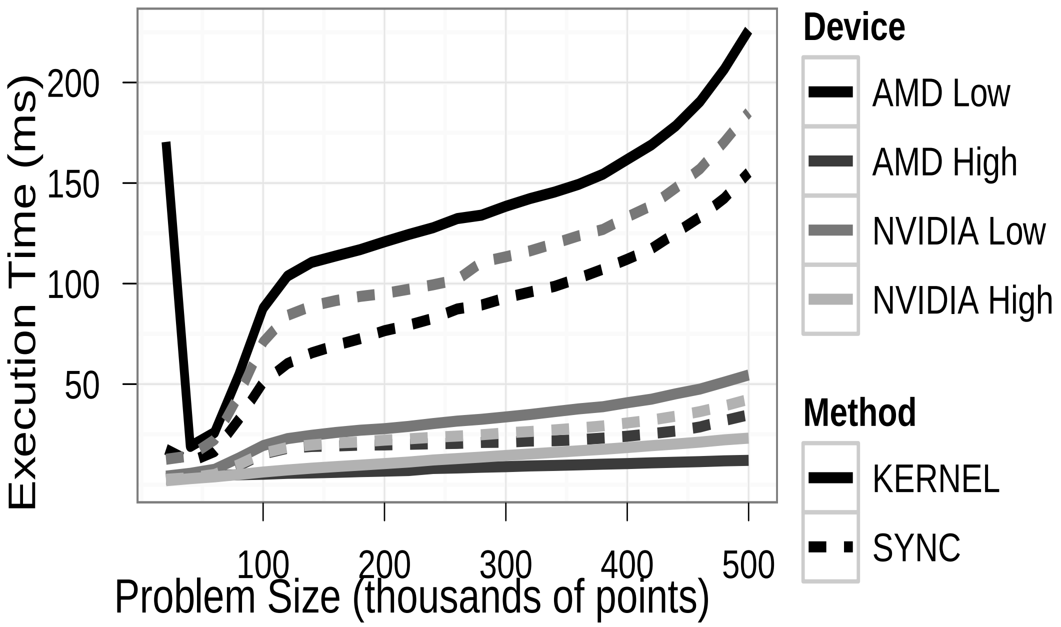


Figure 4.7: Octree performance on a uniformly distributed data set.

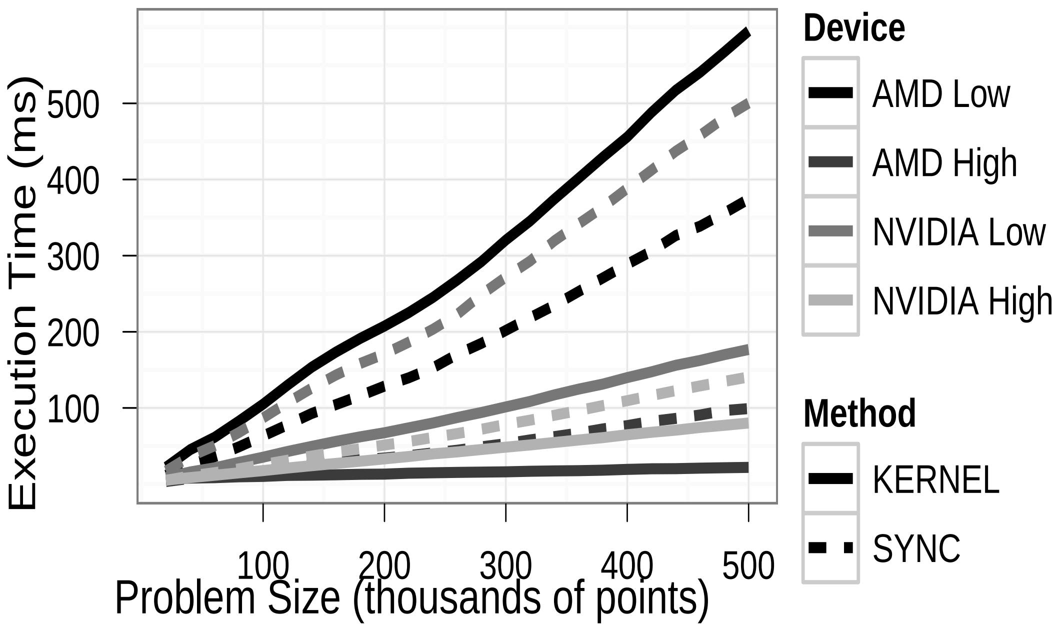


Figure 4.8: Octree performance for a cylindrically-shaped data set.

Chapter 5

Summary and Future Work

In this section we present a summary of our work, including both memory movement techniques and improved synchronization primitives which avoid the costs of contentious atomic memory operations. We also present ideas for future work which use this thesis as a basis.

5.1 Summary

In this section we present a summary of the work completed as part of this thesis, specifically in the characterization and exploitation of memory systems of GPU and APU systems.

Heterogeneous computing has proven to be more than just a fad in computing, but rather has made significant gains in altering the modern computational paradigm. The realization of so many hardware supported threads on Graphics Processing Units (GPUs) make them very attractive for applications which must perform numerous computations in a data-parallel manner. Although the architecture of the GPU allows many applications to achieve instantaneous speedups, many applications are hindered from these improvements by the memory systems of modern GPUs.

This thesis aims to stress the importance of utilizing the underlying system architecture to improve performance for applications which are in some way constrained by the memory system. Using the unique architecture of the Accelerated Processing Unit (APU) we were able to effect large performance gains for data-intensive applications. We were able to achieve a 2.5-fold speedup by using the APU on the VectorAdd application as well as show up to 3-fold performance difference based on the memory movement technique being used. In addition to our work on the APU, we also investigated contentious global memory atomics and their impact on application performance on discrete GPU systems. By leveraging our knowledge we were able to design two novel GPU synchronization primitives, D-Lock and D-Sem, which showed improved performance and scalability on some of the systems we tested.

In general, this work shows the importance of understanding the underlying memory system of the architecture in use. In addition we show the importance of understanding how a given application uses the memory system in order to achieve speedups over naively written codes.

5.2 Future Work

In this section we present future work that can done based upon the work presented in this thesis.

Automated Model for Multiple Devices: The work presented for this thesis looks at improving the performance of data-intensive kernels on the APU, but acknowledges that computationally intensive kernels should still be run on the discrete GPU to leverage the better computational abilities of that system. A system or model could be developed that would automatically predict the best device to run a given kernel based on characteristics about the application, thereby improving the usability of heterogeneous systems by programmers.

Optimization for Multiple Kernel Invocations: Chapter 3 presents a model and performs optimizations for single kernel applications. Multi-kernel applications could achieve best performance running on multiple different devices. A system could be developed which could *a priori* or greedily schedule kernels on whichever device would give them the best performance. However, this performance may not be optimal because of the data transfers that must occur to move computation between devices. Performing this type of analysis can be very beneficial in terms of realistic GPGPU applications.

Shared Memory Synchronization Primitives: With tighter coupling of system memories in newer iterations of the AMD Fusion APU, we hope to extend our work on global synchronization primitives to multiple devices. If the future APUs have a truly shared system memory it may be possible to create synchronization primitives which can work effectively between the CPU and GPU sub-devices. This would increase the ability of collaborative computing between the CPU and GPU on these novel APU systems.

Bibliography

[1] The Top500 Project. http://www.top500.org/, November 2011.

[2] Timo Aila and Samuli Laine. Understanding the Efficiency of Ray Traversal on GPUs. In *Proc. High-Performance Graphics 2009*, 2009.

[3] AMD Corporation. AMD Graphics Cores Next (GCN) Architecture. http://www.amd.com/us/Documents/GCN_Architecture_whitepaper.pdf, June 2012.

[4] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[5] Ramu Anandakrishnan, Tom R.W. Scogland, Andrew T. Fenley, John C. Gordon, Wuchun Feng, and Alexey V. Onufriev. Accelerating Electrostatic Surface Potential Calculation with Multi-Scale Approximation on Graphics Processing Units. *Journal of Molecular Graphics and Modelling*, 28, April 2010.

[6] S. Baghsorkhi, M. Delahaye, S. Patel, W. Gropp, and W. Hwu. An Adaptive Performance Modeling Tool for GPU Architectures. *SIGPLAN Not.*, 45:105–114, January 2010.

[7] P. Boudier and G. Sellers. Memory System on Fusion APUs: The Benefits of Zero Copy. In *AMD Fusion Developer Summit*. AMD, 2011.

[8] Alexander Branover, Denis Foley, and Maurice Steinman. AMD Fusion APU: Llano. *IEEE Micro*, 32:28–37, 2012.

[9] Daniel Cederman and Philippas Tsigas. On Dynamic Load Balancing on Graphics Processors. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '08, pages 57–64, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, and K. Lee, S.and Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IEEE Int'l Symp. on Workload Characterization*, 2009.

[11] M. Daga, A.M. Aji, and Wu chun Feng. On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing. In *Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on*, pages 141 –149, july 2011.

[12] Mayank Daga, Wu-chun Feng, and Thomas Scogland. Towards Accelerating Molecular Modeling via Multi-Scale Approximation on a GPU. In *Proceedings of the 2011 IEEE 1st International Conference on Computational Advances in Bio and Medical Sciences*, ICCABS '11, pages 75–80, Washington, DC, USA, 2011. IEEE Computer Society.

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

[14] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil Computation Optimization and Auto-tuning on State-of-the-art Multicore Architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[15] Wu-chun Feng, Heshan Lin, Thomas Scogland, and Jing Zhang. OpenCL and the 13 Dwarfs: A Work in Progress. In *Proceedings of the third joint WOSP/SIPEW international conference on Performance Engineering*, ICPE '12, pages 291–294, New York, NY, USA, 2012. ACM.

[16] Kirill Garanzha and Charles Loop. Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing. *Computer Graphics Forum*, 29(2):10, May 2010.

[17] Isaac Gelado, John H. Kelm, Shane Ryoo, Steven S. Lumetta, Nacho Navarro, and Wen-mei W. Hwu. CUBA: An Architecture for Efficient CPU/co-processor Data Communication. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 299–308, New York, NY, USA, 2008. ACM.

[18] S.R. Gutta, D. Foley, A. Naini, R. Wasmuth, and D. Cherepacha. A Low-power Integrated x8664 and Graphics Processor for Mobile Computing Devices. In *Int'l Solid-State Circuits Conference Digest of Technical Papers*, Feb. 2011.

[19] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel Prefix Sum (Scan) with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 39, pages 851–876. Addison Wesley, August 2007.

[20] Owen Harrison and John Waldron. Optimising Data Movement Rates for Parallel Processing Applications on Graphics Processors. In *Proceedings of the 25th conference on Proceedings of the 25th IASTED International Multi-Conference: parallel and distributed computing and networks*, PDCN'07, pages 251–256, Anaheim, CA, USA, 2007. ACTA Press.

[21] Tayler H. Hetherington, Timothy G. Rogers, Lisa Hsu, Mike OConnor, and Tor M. Aamodt. Characterizing and Evaluating a Key-value Store Application on Heterogeneous CPU-GPU Systems. *IEEE International Symposium on Performance Analysis of Systems and Software*, 2012.

[22] S. Hong and H. Kim. An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness. *SIGARCH Comput. Archit. News*, 37:152–163, June 2009.

[23] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. Automatic CPU-GPU Communication Management and Optimization. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 142–151, New York, NY, USA, 2011. ACM.

[24] Gary J. Katz and Joseph T. Kider, Jr. All-Pairs Shortest-Paths for Large Graphs on the GPU. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '08, pages 47–55, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[25] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. Modeling GPU-CPU Workloads and Systems. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 31–42, New York, NY, USA, 2010. ACM.

[26] Khronos Group. OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems. `http://www.khronos.org/opencl/`.

[27] Khronos Group. OpenGL Shading Language Specification v 4.2. `http://www.opengl.org/documentation/glsl/`.

[28] Kenneth Lee, Heshan Lin, and Wu-chun Feng. Performance Characterization of Data-intensive Kernels on AMD Fusion Architectures. *Computer Science - Research and Development*, pages 1–10, 2012. 10.1007/s00450-012-0209-1.

[29] Kenneth S. Lee, Heshan Lin, and Wu-chun Feng. Poster: Characterizing the Impact of Memory-access Techniques on AMD Fusion. In *Proceedings of the 2011 companion on High Performance Computing Networking, Storage and Analysis Companion*, SC '11 Companion, pages 75–76, New York, NY, USA, 2011. ACM.

[30] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 451–460, New York, NY, USA, 2010. ACM.

[31] Yang Liu, Wayne Huang, John Johnson, and Sheila Vaidya. GPU Accelerated Smith-Waterman. In *International Conference on Computational Science (4)'06*, pages 188–195, 2006.

[32] NVIDIA Corporation. CUDA C Programming Guide. `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf`.

[33] Martín Pedemonte, Enrique Alba, and Francisco Luna. Bitwise Operations for GPU Implementation of Genetic Algorithms. In *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, GECCO '11, pages 439–446, New York, NY, USA, 2011. ACM.

[34] Kyle Spafford, Jeremy Meredith, and Jeffrey Vetter. Maestro: Data Orchestration and Tuning for OpenCL Devices. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part II*, Euro-Par'10, pages 275–286, Berlin, Heidelberg, 2010. Springer-Verlag.

[35] Kyle L. Spafford, Jeremy S. Meredith, Seyong Lee, Dong Li, Philip C. Roth, and Jeffrey S. Vetter. The Tradeoffs of Fused Memory Hierarchies in Heterogeneous Computing Architectures. In *Proceedings of the 9th conference on Computing Frontiers*, CF '12, pages 103–112, New York, NY, USA, 2012. ACM.

[36] Stanford University Graphics Lab. BrookGPU. `http://graphics.stanford.edu/projects/brookgpu/`.

[37] Jeff A. Stuart and John D. Owens. Efficient Synchronization Primitives for GPUs. *CoRR*, abs/1110.4623, 2011.

[38] Stanley Tzeng, Anjul Patney, and John D. Owens. Task Management for Irregular-Parallel Workloads on the GPU. In Michael Doggett, Samuli Laine, and Warren Hunt, editors, *High Performance Graphics*, pages 29–37. Eurographics Association, 2010.

[39] Vasily Volkov and James W. Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 31:1–31:11, Piscataway, NJ, USA, 2008. IEEE Press.

[40] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU Microarchitecture through Microbenchmarking. In *IEEE Int'l Symp. on Performance Analysis of Systems Software*, March 2010.

[41] Shucai Xiao and Wu-chun Feng. Inter-Block GPU Communication via Fast Barrier Synchronization. In *24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Atlanta, Georgia, USA, April 2010.

[42] Yao Zhang and John D. Owens. A Quantitative Performance Analysis Model for GPU Architectures. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 382–393, Washington, DC, USA, 2011. IEEE Computer Society.

[43] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time KD-tree Construction on Graphics Hardware. *ACM Trans. Graph.*, 27:126:1–126:11, December 2008.