A Data and Contention Aware Approach to Dynamic Scheduling for Heterogeneous Processors

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy (Computer Engineering)

by

Chris Gregg

May 2012

Abstract

Heterogeneous computers, with a multi-core central processing unit (CPU) and one or more many-core graphical processing units (GPUs) capable of running general purpose applications, are becoming standard on the desktop and in cluster and supercomputing platforms. Using programming languages and extensions such as CUDA or OpenCL, application developers can write code that can run on any of the heterogeneous devices available on the system. As these tools mature, contention for device resources will become more prevalent, especially in systems that utilize an application queue. Current application scheduling methods only consider single applications and preferentially schedule each application for a device regardless of other applications in the queue. This degrades overall computational throughput, and leads to device underutilization.

Scheduling application kernels across all heterogeneous components to maximize application throughput is nontrivial and requires the scheduler to have knowledge of the location of data that will be needed for each kernel, the state of the system when each kernel will be launched, and the scheduler be able to predict runtimes for individual applications. Furthermore, a scheduler can utilize historical information about prior kernel run times to further influence its decision.

This dissertation investigates these elements of heterogeneous scheduling and develops a novel methodology for efficiently making dynamic scheduling decisions that maximize computational throughput when multiple applications are run on a heterogeneous computer. This dissertation describes an innovative taxonomy for describing data transfer requirements for a heterogeneous application, and it also shows that dynamic scheduling for heterogeneous computers benefits from knowledge about system state information, including data locality and contention among running processes. Finally, this dissertation investigates a novel technique to increase GPU throughput by running concurrent kernels on a device, based upon their orthogonal use of device resources.

Approval Sheet

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Engineering)

[illegible]

Chris Gregg

This dissertation has been read and approved by the Examining Committee:

[illegible]

Kim Hazelwood, Adviser

[illegible]

Kevin Skadron, Committee Chair

[illegible]

Marty Humphrey

[illegible]

Benton Calhoun

Norman Rubin

Norman Rubin

Accepted for the School of Engineering and Applied Science:

[illegible]

James H. Aylor, Dean, School of Engineering and Applied Science

May 2012

Acknowledgments

As every Ph.D. student knows, his or her advisor is the most important person on the road that leads to a completed dissertation. My advisor, Kim Hazelwood, is no exception, and without her guidance, suggestions, ideas, opinions, words of wisdom, and, above all, her continual prodding to make my work better, I would still be many miles from my dissertation, and probably out of gas. I have learned a great deal about the world of academia from Kim, and she has given me many opportunities to make the most of graduate school, not the least of which was the opportunity to fill in teaching her Computer Architecture course for a semester, which will undoubtedly help my prospects to continue with my teaching career.

Virtually all of this dissertation has, in other forms, passed by the experienced and perceptive eyes of Kevin Skadron, and his guidance, insight, support, and advice about heterogeneous computing and its idiosyncrasies was invaluable to the direction I took this work. Additionally, every experiment in this dissertation was at least partially completed on computers and equipment from Kevin's LAVA Lab, and without those resources this work would not have a single result.

Other professors who have given me outstanding guidance have been Joanne Dugan, Marty Humphrey, Ben Calhoun, and Wes Weimer. I am also grateful to Norm Rubin for the opportunity to work for a summer at AMD, and also to Perhaad Mistry, my deskmate and collaborator at AMD. In the Computer Engineering office at UVA, Natalie Edwards made sure that I had all of my administrative paperwork complete, and I am thankful for her behind-the-scenes support.

Among my graduate school colleagues, Michael Boyer stands out as the most important source for advice, guidance, and friendship throughout my graduate school career. As a fellow computer architect and collaborator (and much more experienced in all things related to GPUs), Michael is always available for brainstorming sessions, and I unfailingly end up coming away from those discussions with a better way forward because of his ideas and insight. I also need to thank Marisabel Guevara, another former colleague, as well – Marisabel introduced me to heterogeneous computing and her suggestions germinated this dissertation.

I could not have completed the last few stages of this work without Jonathan Dorn's help, coding expertise, and intricate understanding of GPUs and in particular, OpenCL. I drove him mad with my own horrible coding practices, and I hope he forgives me for that. Others who have also helped with particular technical advice include Shuai Che, Ryan Layer, Michelle McDaniel, Luther Tychonievich, Dan Upton, Jason Mars, and Duane Merrill, and I thank them for their help.

I must thank Ray Buse for his friendship and advice over the past few years. I learned more about the field of computer science from Ray than from anyone else, and whenever I wondered why I was subjecting myself to this field, all I would have to do is talk to Ray for a few minutes and he would remind me again why it is so fascinating and fun.

There were a number of non-UVA friends who have supported me through the past four years, as well; without their encouragement, I don't think I would have made it through the more trying times: Derek Seabury, Andrea deManbey, Payton Smith, Stacy Kissel, Jessica Webster, Marci Simon, Patricia Burt, Ally Mizoguchi, Jenn Daly, and Erica Dreisbach.

Most importantly, I must thank my family for their support, especially because my graduate career spanned a number of difficult periods for my family. My mother, sister, brother-in-law, and my Aunt Liz and Aunt Patrice were particularly supportive with regard to both my school work and with family issues, and I thank them all.

Finally, I would like to thank a the National Science Foundation, Semiconductor Research Corporation, and Microsoft for a number of grants that supported my research throughout my graduate career. Specifically, this work was supported by NSF grants CNS-0747273, CSR-0916908, and CNS-0964627, and SRC GRC task number 1790.001.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Graphics Processing Units (GPUs) that are capable of executing general purpose code have become ubiquitous on both the desktop and in cluster and supercomputing platforms. Because GPUs contain many processing cores, they are also able to run parallel code efficiently, and for certain applications can deliver tremendous performance increases over equivalent code written for a general purpose processor. By leveraging one or more GPUs to perform a subset of tasks in a heterogeneous computer consisting of CPUs and GPUs, it is possible to increase the overall computational throughput of the machine. If some of these tasks can be run on either the CPU or GPU, it becomes necessary to make a scheduling decision about when and where the tasks will be run in order to maximize this throughput.

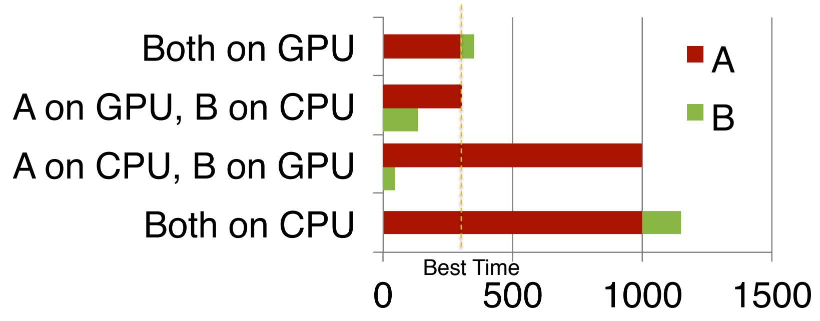
Parallel programming languages targeted for GPUs, such as CUDA [NVIDIA, 2011b] and OpenCL [Stone et al., 2010], have started to gain a following among application developers who want to leverage general purpose GPU computation. Both languages require the application developers to explicitly schedule *kernels* (the computation performed on the device, not including data transfer) onto a particular processing unit. Each application developer must determine when to schedule kernels for the devices on a system, and there is no built-in coordination between applications if multiple applications wish to schedule kernels on a particular device. As more application developers write programs that run on GPUs, contention for the devices will increase, due to general purpose code and also due to graphics code that the GPU handles as its main function. The need for a dynamic scheduler to handle this increased contention will continue to grow in importance.

Ideally, a computer system should be able to utilize all of its processors, efficiently delivering work to the processor that is best suited to complete the work as quickly as possible. Modern operating systems for commodity and scientific computing have the ability to run multiple applications at once, and OS schedulers are already able to partition work across a number of homogeneous cores on a multi-core CPU. However, there are many challenges associated with extending those schedulers to include a GPU, which has different strengths and weaknesses than a CPU, and may or may not be the best place to run a section of code. Incompatible instruction set architectures between CPUs and GPUs further complicate running the same code on both devices. Even when using a programming language such as OpenCL that can target both architectures, it is beneficial to tune code specifically for one device. In addition, because discrete CPUs and GPUs have independent memories, necessary data must be moved between the two processors, and we will show that a scheduler must also take this data movement overhead into consideration.

Given the aforementioned factors, scheduling decisions change drastically, and must take into account the differences in application runtimes between the CPU and GPU, the location of the data that will be used in the computation, and which device (either the CPU or GPU) is available to complete the work. In other words, the scheduler must have a certain amount of metadata associated with the application, about the processors, and must have knowledge of the environment in order to make a worthwhile decision about where to run the task. GPU scheduling methods prior to the work in this dissertation assume that GPUs are unloaded when queuing work for a device. If the device is currently processing another task, the new work must wait to run, even if other devices could be utilized. This leads to bottlenecks on the GPU, and underutilization of the CPU, and this dissertation addresses these issues by considering the metadata available.

This dissertation shows that ignoring system state information, including data locality and contention among running processes leads to incorrect scheduling decisions that are detrimental to maximum computational throughput. By utilizing historical runtime data for heterogeneous kernels and by making use of how those kernels interact with each other while running concurrently, we can dynamically determine a schedule that provides better overall throughput than current scheduling methods.

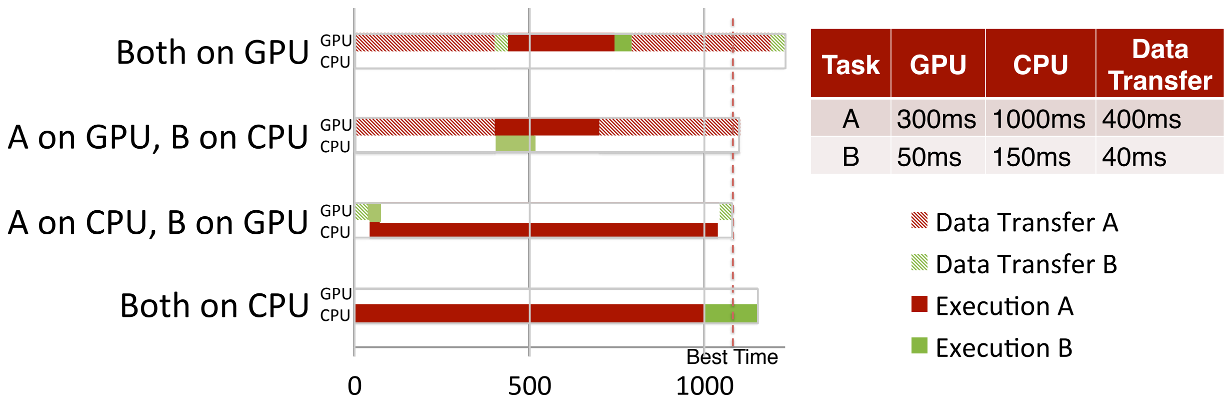
1.1 Scheduling Decision Examples



Task	GPU	CPU
A	300ms	1000ms
B	50ms	150ms

Figure 1.1: An example of a scheduling decision that produces the maximum throughput by running one task on a slower device. In this system, both tasks are capable of running on either the CPU or GPU, and both tasks run faster on the GPU. A naïve placement of both tasks on the fastest device produces a slower overall runtime.

Naïve scheduling has clear weaknesses, particularly when only one device is targeted for all applications. For example, Figure 1.1 demonstrates a simple scheduling example by showing two tasks that can run on either a CPU or a GPU in a heterogeneous computer. Both tasks run faster on the GPU, but scheduling both for the GPU results in a slower overall finish time than the optimal scheduling of task A on the GPU and task B on the CPU. Although task B takes more time on the CPU, the utilization of both devices in parallel leads to a better overall throughput. This dissertation shows that the current technique of making scheduling decisions without considering queue wait times, or when applications will finish on a given device, leads to suboptimal throughput for multiple applications that have the ability to run on either device.



Task	GPU	CPU	Data Transfer
A	300ms	1000ms	400ms
B	50ms	150ms	40ms

Figure 1.2: An example of a scheduling decision that includes data transfer time, and when the data initially resides in CPU memory. Because the transfer time for task A is significant, the best scheduling decision is to run task A on the CPU and task B on the GPU.

Figure 1.2 shows an example of a case where the optimal scheduling decision necessitates knowing not only the execution time on each device, but where the data resides on the machine. Discrete CPUs and GPUs have independent memory, and data must be explicitly transferred between the two devices across the PCI-Express bus. In figure 1.2, the data starts in CPU memory and is transferred to the GPU for computation and then returned to the CPU after the computation completes. Given the execution and data transfer times, the best schedule is to run task A on the CPU and task B on the GPU, even though the task runs significantly slower on the CPU than on the GPU. Prior work that has looked at GPU speedup and scheduling [Lee et al., 2010b; Topcuoglu et al., 1999] has ignored data transfer times, leading to poor overall runtime comparison and suboptimal schedules.

Figures 1.1 and 1.2 demonstrate the need for sophisticated scheduling decisions based on at least three factors:

1. Knowledge of task runtimes on individual heterogeneous components. Runtime predictions must be accurate enough and must be made quickly.

2. The workload currently executing on each device. By managing contention for processing units, overall computational throughput can be increased. For many-core GPUs, maximum utilization of individual cores can be increased by scheduling more than one task at a time on the GPU.

3. The location of data relative to the processing units. A scheduler must take into account data transfer overhead when considering where to run a task.

These items provided the initial motivation for the work in this dissertation, leading to the contribution of a scheduling methodology for heterogeneous computers that includes the essential factors for maximizing the throughput for a set of tasks in a realistic execution scenario.

1.2 Contributions of this Work

The thesis of this work is that dynamically scheduling compute kernels for a heterogeneous computer is essential for high compute throughput and high device utilization. Historical runtime data and system state information must be used in a dynamic scheduling algorithm, and will provide better overall throughput than current scheduling methods. A dynamic scheduler that uses this information will produce schedules that approach the runtime of an oracle scheduler.

This dissertation investigates the components necessary for a dynamic heterogeneous scheduler for computers that contain at least one CPU and one GPU. It addresses data locality for scheduling across processors with independent memories, runtime prediction for heterogeneous kernels, and presents a general scheduling algorithm based on the results of that work. Additionally, the dissertation investigates co-scheduling of heterogeneous kernels on a single device, leveraging the fact that individual kernels tend to underutilize different device resources. Finally, this dissertation shows that running two orthogonal kernels together can improve aggregate throughput for a single device.

1.2.1 Analysis of Data Locality for Discrete Devices

We perform a study of the data transfer times for a set of heterogeneous benchmark kernels across a number of devices. We show that the time to transfer data can contribute significantly to the overall runtime of a kernel and that data transfer times should not be ignored when comparing runtimes between heterogeneous devices. We also demonstrate that knowing how data will be consumed after a kernel finishes can also provide insight into total device throughput. Chapter 3 investigates:

- **The Necessity of Including Data Transfer Times in Runtime Comparisons:** We show that the time to transfer data across a PCI-Express bus can significantly impact overall runtimes, and kernels that show better performance on a device are affected the most.

- **Data Transfer Patterns:** We provide a taxonomy for describing data locality patterns for a heterogeneous kernel, and recommendations for how to use the taxonomy for application comparisons when data must be transferred between device memories.

1.2.2 Runtime Prediction for Heterogeneous Devices

Because of the runtime differences between heterogeneous devices, a scheduler must have access to runtime predictions or models in order to make effective scheduling decisions. We explore runtime prediction models based on a history database of previous runtimes. Chapter 4 investigates:

- **Using Historical Runtime Data for Future Runtime Predictions:** We demonstrate the feasibility of collecting runtime data and utilizing it to make future runtime predictions. We provide a novel method for keeping database sizes small while retaining enough information for fast and precise predictions.

- **Runtime Prediction Models:** We introduce a novel hybrid kernel runtime prediction model based on historical runtime data that provides fast and precise prediction suitable for a dynamic heterogeneous scheduler.

1.2.3 Contention-Aware Dynamic Heterogeneous Scheduling

Using the results from Chapters 3 and 4, we examine heterogeneous scheduling that utilizes data locality and is aware of device contention. Chapter 5 investigates:

- **Necessary Heterogeneous Scheduling Information:** We provide a characterization of the information necessary to inform heterogeneous scheduling decisions, including data locality and device contention.

- **A Dynamic Heterogeneous Scheduling Algorithm:** We introduce a novel heterogeneous scheduling algorithm that utilizes runtime prediction and system state information. We test the algorithm on a set of OpenCL benchmarks, and we show that it produces schedules that are within 12% of an oracle schedule. We also compare the scheduling results to four alternate scheduling algorithms and show that the dynamic scheduling algorithm is superior in each case.

- **Dynamic Scheduling With and Without Data Transfer Predictions:** We compare implementations of our dynamic scheduling algorithm with and without utilizing data transfer predictions, and we show that by including those predictions, the schedules produced are, on average, 40% faster. This further validates the results from Chapter 3.

- **System Device Utilization:** We show that dynamically scheduling work across multiple devices increases device utilization compared to other scheduling algorithms. Our dynamic scheduler can achieve over 95% device utilization across both the CPU and GPU. We also show that greater device utilization improves aggregate throughput across all devices.

1.2.4 Co-scheduling Heterogeneous Kernels on a Single GPU

We analyze GPU kernel operation and show that kernels can demonstrate a range of memory-bound and compute-bound behaviors. Using this insight, we show that orthogonally bound kernels can be run concurrently on a device and demonstrate improved aggregate throughput compared to running the kernels independently. Chapter 6 investigates:

- **Orthogonally Bound Kernels:** An analysis of contention and symbiotic behavior between concurrent GPU kernels. We demonstrate that individual kernels can under-utilize GPU resources, but when a memory-bound kernel is run concurrently with a compute-bound kernel, overall throughput can be improved, and greater device utilization occurs. We show improved computational throughput when concurrent kernels are scheduled intelligently, and degraded throughput with naïve scheduling.
- **Concurrent Scheduling on Real Devices:** We provide a novel prototype OpenCL kernel scheduler that demonstrates concurrent scheduling at workgroup granularity. The scheduler has two modes of operation. In the first, the scheduler proportionally allocates GPU resources to two kernels, at workgroup granularity. The second mode allocates workgroups of individual kernels to GPU compute devices as those devices become available ("work-stealing"). We show that the work-stealing scheduler produces schedules that approximate the throughput for the best proportionally allocated schedules.

These contributions provide clear evidence demonstrating the need for and benefits of dynamic heterogeneous scheduling on a CPU/GPU computing platform. The rest of this dissertation is laid out as follows. Chapter 2 provides the background for the work, including the current state of General Purpose Graphics Processing Unit (GPGPU) computing and a prediction of how GPGPU computing will evolve in the next decade. The chapter will also discuss the differences in CPU and GPU architectures, and describes the difficulties in simply extending current Operating System schedulers to include GPUs. Chapters 3–6 provide the nucleus of the dissertation and the contributions discussed above. We review related work in Chapter 7, including previously described heterogeneous schedulers. Chapter 8 concludes the dissertation and details a proposal for future work integrating heterogeneous scheduling into an operating system at the kernel level.

1.3 Citations to Previous Publications

Large portions of this dissertation appeared in the following publications:

C. Gregg, J. Dorn, K. Skadron, K. Hazelwood. "Fine-Grained Resource Sharing for Concurrent GPGPU Kernels," 4th USENIX Workshop on Hot Topics in Parallelism (HotPar '12). June 2012.

C. Gregg, M. Boyer, K. Hazelwood, K. Skadron. "Dynamic Heterogeneous Scheduling Decisions Using Historical Runtime Data," Proceedings of the 2nd Workshop on Applications for Multi- and Many-Core Processors. San Jose, CA, June 2011.

C. Gregg and K. Hazelwood. "Where Is the Data? Why You Cannot Debate CPU vs. GPU Performance Without the Answer," International Symposium on Performance Analysis of Systems and Software (ISPASS). Austin, TX. April 2011.

C. Gregg, J. Brantley, and K. Hazelwood. "Contention-Aware Scheduling of Parallel Code for Heterogeneous Systems," 2nd USENIX Workshop on Hot Topics in Parallelism (HotPar '10). June 2010.

M. Guevara, C. Gregg, K. Hazelwood, and K. Skadron, "Enabling Task Parallelism in the CUDA Scheduler," in Programming Models and Emerging Architectures Workshop (Parallel Architectures and Compilation Techniques Conference, 2009).

Chapter 2

Background

2.1 The Current State of GPGPU and Heterogeneous Computing

General-purpose computing on graphics processing units (GPGPU) describes the use of GPUs to perform conventional computing that is not specifically targeted for output to a graphical display. When GPGPU programming was originally conceived*, only graphics APIs were available and exposed to programmers, and in order to solve general purpose problems using a GPU, they had to be framed in terms of graphics-specific functions and memory models [Owens et al., 2008]. This was a tedious, low-level process, and GPGPU programming was generally limited to researchers who were curious about the possibility of large speedups and who had the time to dedicate to translating their code into a non-intuitive framework.

The situation changed significantly when GPU vendors started producing drivers and APIs for the sole purpose of writing general purpose code. ATI (now AMD) created the *Close To Metal* (CTM) low-level programming interface in 2006, and this became commercially available (and with higher level tools) the following year as the AMD Stream SDK [AMD, 2012a].

*GPGPU computing can be nominally traced back to 1978 with the Ikonas Graphics Systems [England, 1978], but GPGPU programming on modern GPUs began around 1999–2000 when NVIDIA released the GeForce 256 [Wu and Liu, 2008].

NVIDIA released *Compute Unified Device Architecture* (CUDA) in 2008, which enables access to a low-level virtual instruction set and to the GPU memory system. [NVIDIA, 2011b] NVIDIA also provided both C and Fortran compilers to allow high level programming through CUDA. Also in 2008, the *OpenCL* specification was released, and OpenCL provides a C-based language for parallel programming on both GPUs and multicore CPUs [Stone et al., 2010].

Because languages such as OpenCL are targeted towards both GPUs and multicore CPUs [*], the term *Heterogeneous Computing* (instead of *GPGPU*) has come into favor when discussing applications written in these languages. We will be using this term for the remainder of this dissertation.

2.1.1 Kernel Scheduling

Heterogeneous computing applications have two distinct parts: the coordination code, which runs as a regular process on a CPU, and the *kernels*, which run on the GPU or on the CPU. When running heterogeneous code on a GPU, the coordination code runs on the CPU and the kernels run on the GPU, and when running heterogeneous code on a multicore CPU, both the coordination code and the kernels run on the CPU, in different processes.

With OpenCL and CUDA, all scheduling of kernels and all memory transfers are explicitly handled by the individual application developer who writes heterogeneous code. Unless application development is done by a single developer or by coordinating developers, applications do not communicate with each other, and kernels are launched onto a device when the coordinating code for each application calls a library function. Kernels go into a first-in-first-out (FIFO) queue for the device, and only one kernel is run on a device at a time. CUDA has a rudimentary ability to launch multiple kernels concurrently, but the mechanics of this process have to be carefully coordinated by an application developer [NVIDIA, 2011b].

If a computing platform has more than one device that is capable of running heterogeneous kernels, application developers must also explicitly decide which device to run the kernels

[*]Although CUDA is officially limited to running on NVIDIA GPUs, there have been research efforts that have aimed at translating CUDA for CPUs [Stratton et al., 2008].

on. When writing applications, developers do not know the state of the system, and may not know what specific devices will be available at runtime. Therefore, they cannot predict whether a device is currently running a kernel, or whether there are other kernels queued up behind one that is currently running, or whether data transfer overhead will considerable affect overall runtime. Developers do have the ability to query the system to determine what devices are available to run heterogeneous code, but the decision about which device to launch kernels on is currently determined prior to runtime, or logic must be built into the individual application that makes this determination at runtime. Again, unless two or more applications are created in tandem, there is no dynamic method for scheduling the kernels except through the FIFO queue.

2.2 The Future of Heterogeneous Computing

One of the goals of this dissertation is to provide a motivation for developing a heterogeneous scheduler that considers all applications that have heterogeneous kernels. This motivation is based on the idea that as more developers write heterogeneous code, there will be more contention for heterogeneous devices in a system. Current static scheduling techniques based on device driver queues does not provide any method for managing contention between independent applications.

Figure 2.1 shows the number of publications listed in Google Scholar [2012] that refer to CUDA or OpenCL in the context of programming, showing a linearly increasing trend. As more applications are written to take advantage of heterogeneous resources, dynamic scheduling for kernels across the available devices will be necessary to avoid bottlenecks and to fully utilize the devices. Applications written by different developers will have to share access to preferred devices, but if overall throughput for all applications is the goal, it will not matter which device an application kernel runs on as long as it finishes sooner than it would have on a different device.

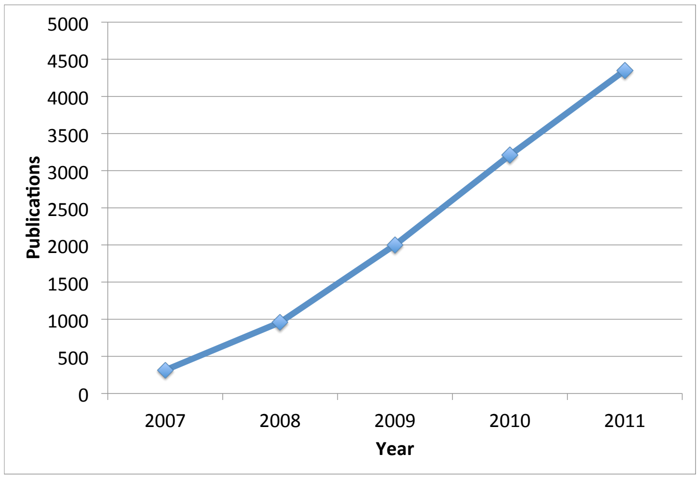


Figure 2.1: The increasing trend in the number of publications per year that refer to “programming AND (CUDA OR OpenCL)”. Source: Google Scholar [Google, 2012]

The obvious place to build a heterogeneous scheduler that handles multiple applications running simultaneously is the operating system. The OS already handles homogeneous scheduling tasks, and adding the ability to schedule for heterogeneous devices is an eventual progression. This dissertation investigates a number of features that would enhance an OS scheduler that included support for heterogeneous processors.

Although discrete CPUs and GPUs will continue to be available in the foreseeable future, hardware vendors have also begun building heterogeneous processors onto a single chip, with both devices sharing a single main memory [AMD, 2010], and both are capable of running general purpose code. The integrated main memory allows kernels to run on either device without needing to transfer data. For this reason, data locality would be less important, however other memory-specific issues would then arise, such as increased contention for memory resources such as memory controllers and caches.

2.3 Many-core GPU Architectures

GPUs are designed to efficiently handle the graphics pipeline, which consists of a large number of graphics elements that can be altered and manipulated by applying the same function to each element. Because each pixel can be independently rendered, the elements can be processed in a highly parallel fashion. In order to leverage the parallel nature of the data, GPUs have a large number of processing cores with tremendous floating point throughput capabilities and fast memory bandwidth. In addition, GPUs partition the cores into groups of cores, each with their own shared memory. Figure 2.2 shows NVIDIA's streaming multiprocessor model, which is repeated multiple times on a single chip.

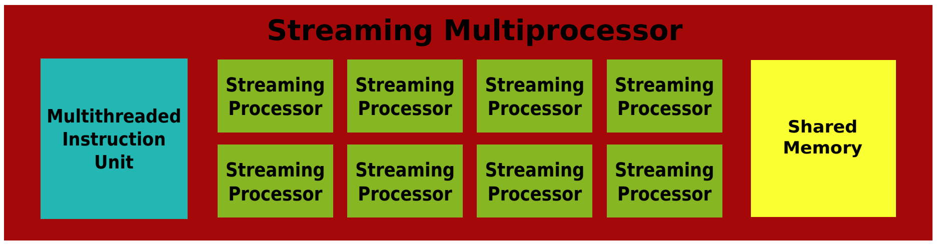


Figure 2.2: The NVIDIA GPU Streaming Multiprocessor Architecture. Each set of processing cores can access the same shared memory. The Multithreaded Instruction Unit handles thread launching for the multiprocessor.

GPUs fall under the Single Instruction, Multiple Data (SIMD) classification of Flynn's Taxonomy [Flynn, 1966], and successful general purpose code that is run on GPUs follows this model for its algorithms. In the SIMD model, the same instruction is executed for a stream of data, and because the graphics pipeline necessitates applying functions over a stream of pixel data, the model fits nicely. Data parallel algorithms are common in scientific computing, and scientific applications have led the push for GPGPU computing across a wider variety of computing systems.

Because of the vast amount of data that potentially streams through the processors on a GPU, GPUs have a wide and fast main memory bus. The shared memory on streaming multiprocessors is even faster. This is in contrast to the much slower PCI-Express (PCI-E) bus that allows data transfer between CPU and GPU memory, and this leads to the memory transfer bottleneck that will be discussed in chapter 3.

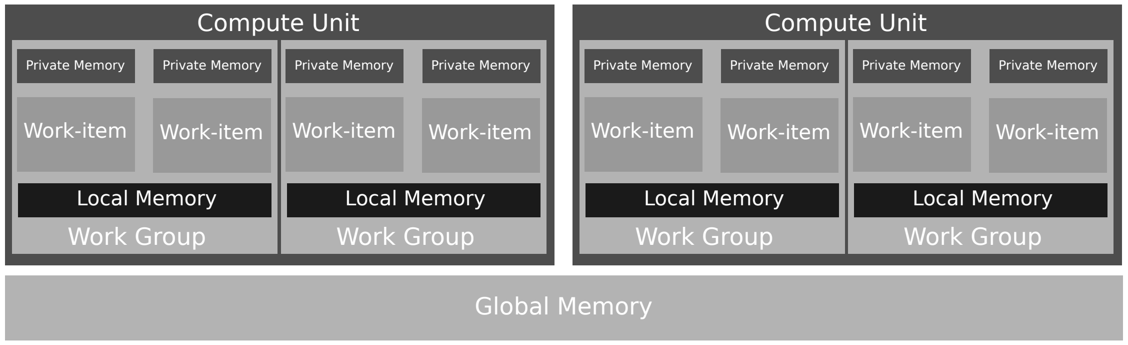


Figure 2.3: The abstract OpenCL programming model and the GPU memory model.

2.4 OpenCL and CUDA Terminology

In this disseration we will be discussing specific OpenCL and CUDA terminology, and this section will provide a brief overview of the terms we will use. OpenCL and CUDA generally have different terms for similar device components and language models, and a more thorough mapping can be found in other references, such as NVIDIA's *OpenCL Optimization Guide* [NVIDIA, 2012c]. Parentheses in this section denote whether a term is specific to OpenCL or CUDA, and a lack of parenthesis means the term is shared between the two languages. In general we will use the OpenCL terminology if we are discussing an idea generic to both OpenCL and CUDA.

Figure 2.3 shows the abstract OpenCL programming model and the GPU memory model; this section describes the former, and the next section describes the latter. An OpenCL or CUDA computational device (e.g., a CPU or GPU) is referred to generically as a *device*. Devices can be discrete or integrated, as in the case of AMD's Fusion architecture. Discrete devices contain independent global memory, while integrated devices share global memory. In the latter case, data is explicitly transferred between devices in order for each device to perform calculations on that data, and on current hardware the transfer takes place via the PCI-Express bus.

Each device contains multiple *compute units* (OpenCL) or *multiprocessors* (CUDA) that perform the computation. All compute units share access to global memory, but devices also have local (OpenCL) or shared (CUDA) memory that is allocated on a per-compute unit basis and cannot be shared between compute units.

Programs that run on a device are called *kernels*. Kernels are launched from a CPU thread, and kernels are managed by a device driver, generally provided by device vendors (e.g., AMD, NVIDIA, and Intel). The device driver also handles data transfer between devices, and in OpenCL the data is encapsulated into *buffers* that are passed during the data transfer.

Each instance of a kernel is called a *workitem* (OpenCL) or *thread* (CUDA), and each workitem executes the same instructions on different parts of the kernel data. Workitems each have a unique *global ID*, and workitems are organized into *workgroups* that can be synchronized by the use of a *barrier*. Each workitem in a workgroup shares a *workgroup ID*, as well, and workitems have unique *local IDs* in their workgroups. Workgroups share access to local memory on the compute unit on which they reside, although workgroups that reside on the same compute unit cannot share local memory with each other.

In both CUDA and OpenCL, applications have the ability to query the system to determine what devices are available and what features they support. Additionally, applications can receive information about kernel runtimes and buffer transfer times through device driver function calls.

2.5 The GPU Memory Model

Discrete GPUs contain multiple levels of memory, with global memory available to all workitems, local memory available to individual workgroups, and private memory available to individual workitems. Global memory can contain multiple gigabytes of system RAM, and private memory is allocated from system RAM as well. Local memory is limited to tens of kilobytes, and allocated on a per-workgroup basis. Chapter 6 discusses the problems with running multiple kernels concurrently with such a limited amount of local memory.

Figure 2.3 shows how the memory fits into the OpenCL model. Local memory can be up to $5x$ faster than global memory, and accessing global memory is up to $10x$ faster than transferring data across the PCI-E bus. Chapter 3 discusses the importance of limiting the amount of transfer between devices, particularly because of the amount of transfer time relative to kernel runtimes. Because PCI-E transfer time is so slow relative to onboard GPU memory accesses, the location of the data becomes important.

While the OpenCL programming model should remain the same, differences arise in the actual GPU memory model for integrated CPU/GPU devices, such as AMD's Fusion line of heterogeneous processors. With an integrated model, there is generally one main DRAM memory system that can be used by both the CPU and the GPU. Because the GPU relies on high throughput access to memory for massively multithreaded code, GPU designers may provide different types of busses from each device. In the case of AMD's Fusion, there is a non-coherent but extremely fast bus (i.e., it can saturate the available bandwidth) that can be used by the GPU for local memory, and separate bus that can be used for coherent access. There is also a separate bus for the CPU to access memory, as well.

Because of the vast differences in memory models between discrete and integrated heterogeneous devices, this dissertation considers only the discrete case.

2.6 Scheduling for CPUs and GPUs

As discussed in chapter 1, the operating system handles CPU scheduling, and individual applications send work to GPUs through GPU vendor device drivers that enforce a strict in-order kernel queue. Because of this separation of scheduling duties, no single scheduling entity has a view of processor utilization for all devices. The operating system does not make any scheduling decisions for the GPU; rather, the programmer for each kernel decides whether to run that kernel on the CPU or GPU. There are a number of factors that make automating that decision difficult, and this dissertation investigates those elements. It is not simply a matter of updating the OS scheduling algorithms to use the GPU, and data transfer between devices, much different device runtimes, and the inability to preempt work that has started on the GPU all play important roles in the decision.

With an increased number of applications trying to use GPU resources, a mismatch can occur where some devices are under-utilized, and others are over-utilized. For example, if multiple applications running on the CPU attempt to run a kernel on the GPU simultaneously and also wait until kernel completion before continuing on the CPU, all but the first into the queue will be blocked, and CPU utilization will drop significantly.

Operating system schedulers are able to partition independent work across multiple CPU cores with the simplifying assumption that homogeneous cores will have equal runtimes, and therefore the cores will behave identically. The scheduler does not need to know absolute runtimes for complete applications in order to schedule work, and ensuring high core utilization and load balancing is a matter of distributing processes to each core depending on the current load on individual cores. More importantly, processes can be migrated between cores as they run. The scheduler regularly checks the load on each core, and migrates processes as necessary to maintain high overall core utilization. This can have negative ramifications for cached data—if a process that has cached data on one core is moved to another, the data needs to be migrated as well—but device utilization remains high.

Enhancing an operating system scheduler to include both CPUs and GPUs would necessitate solving a number of challenges, and this dissertation addresses those challenges in detail. The primary difficulty arises because GPUs do not have the ability to interrupt or migrate processes – once a kernel begins, it runs until completion on the device. Whereas CPU cores can be monitored to assess their load, GPUs are either running a single kernel or not, and fine-grained thread switching between kernels is not natively possible with current GPU architectures. Therefore, in order for an OS scheduler to attempt load balancing between CPUs and GPUs, there must be some knowledge or prediction of the runtime for individual kernels on a device. We analyze the problem of runtime prediction in chapter 4, and we demonstrate a scheduling algorithm that utilizes these predictions in chapter 5. In chapter 6 we analyze some solutions for running multiple kernels on GPUs concurrently.

Even if it becomes possible to easily migrate tasks between a CPU and a GPU, data migration is still a major concern. As already discussed, transferring data between CPU and GPU memories is bandwidth limited, and can incur large overheads on top of kernel runtime. Chapter 3 addresses the data transfer overhead problem, but an operating system scheduler that depended on process migration in a CPU-GPU system would need to ensure data was also migrated between devices (or left in place on both), and either choice would add considerable overhead.

2.7 Characteristic Heterogeneous Workloads

This dissertation focuses on workloads consisting of applications with kernels that can be executed on the CPUs and GPUs in a heterogeneous computing system. Individual applications do not have dependencies on each other, and have no knowledge of each other (i.e., they could have been independently written by different developers). The dynamic scheduler discussed in this dissertation places kernels into a queue and attempts to produce the greatest throughput for all applications without unfairly penalizing an individual application, either by starvation (i.e., preempting an application indefinitely), or by scheduling a kernel to finish later than it would have if it was run on an alternate device. The notion of scheduling fairness is described in detail in Chapter 5.

We envision the types of workloads described above to emerge in both single-user desktop computing and in multiple-user cluster computing where the cluster nodes are comprised of heterogeneous CPU/GPU computers. A number of such clusters are currently in service [Kindratenko et al., 2009; Morgan, 2012], and because individual applications on such devices are generally independent of each other, the scheduling methodology we describe in this dissertation can be used to increase throughput in such clusters.

Chapter 3

Where is the Data? Debating CPU vs. GPU Performance

Numerous papers have demonstrated the impressive computing power of GPUs, and speedups of 1000x across a number of different types of applications have been reported over similar algorithms written for CPUs (for example Che et al. [2008] and Ryoo et al. [2008]). However, there is an ongoing discussion about the quantitative speedup that GPUs can provide over CPUs and whether real applications truly can benefit by two or more orders of magnitude improvement by utilizing GPU computing. Lee et al. [2010b] provide a compelling argument that fair comparisons are necessary; CPU and GPU performance evaluations are only valid when both have been fully optimized for their respective platforms, taking into consideration specific architectural features for each device and optimizing the kernels to take advantage of these as much as possible. We will show that even this argument lacks an important factor: the location of the data that is being processed before, during, and after kernel execution. As we show in this chapter, it is not simply the speed at which a certain algorithm processes data that is important, but where the data is prior to running on a device, and where it will be used after the kernel completes. GPU kernels are fast, and in many cases kernels are completed in the tens of milliseconds time frame. With short kernel runtimes, the movement of data to and from the GPU becomes relevant, both because of the amount of data that needs to be moved, and because of the inherent bottleneck from the interconnect between the CPU and GPU. Without acknowledging (or rationalizing) the time for these data transfers, comparisons do not provide a true representation of the real speedup provided by a device. Our objective in this chapter is not to challenge the benefits of using GPUs to accelerate computational work; rather, we would like to emphasize the importance of including data transfer overhead when making comparisons to CPU applications that have the data accessible on-chip.

It is one thing to acknowledge that for most GPU kernels, data must be moved onto the device prior to being used by the kernel, but it is more important to recognize that calculations performed on data are only worthwhile when the results of those calculations are further utilized. For example, if an array of integers has been sorted, the newly sorted array is useful only if the data will be analyzed by a follow-on function or algorithm. Unless the data will be used by another kernel on the GPU, some data must be returned to the CPU via the interconnect, whether it is the full data set, or a result based on the data (e.g., the result of summing an array of integers). Disregarding this aspect when reporting GPU kernel performance is deceptive, until such a time as there are shared memory CPU/GPU systems and there is no need to transfer data across a device-to-device interconnect for further processing or output.

We have two goals in this chapter. The first is to demonstrate the necessity of reporting memory transfer overhead costs for CPU/GPU performance comparisons, and to show that many performance results have been misleading because they have disregarded this information. We have run benchmarks on eleven diverse GPU kernels and have included timing data for CPU/GPU memory transfers. Our results show that data transfer time can be as significant as kernel runtime, and that the faster the GPU, the more important it is to consider the data transfer overhead. The latter observation demonstrates the increasing importance of our findings, given the expected increases in GPU performance over time.

Many of the benchmarks transfer a significant amount of data onto the GPU before the kernel begins, and a transformed data set back to the CPU after the kernel completes. In some cases, however, a large amount of data is transferred onto the GPU but only a small amount of data is returned to the CPU. For example, the search kernel transfers a large text string to the GPU but only returns an integer value of the location of a found string. When the one-way transfer time is taken into account, the search kernel takes 2x longer than the GPU processing time, on average. In other cases, data does not need to be transferred to the GPU but a large amount of data is transferred back from the GPU. For example, the Mersenne Twister benchmark generates pseudorandom numbers on the device with a single integer as the only input data, but if the results are brought back to the CPU, this adds an average time of 7x over the GPU processing time alone. If instead the results remain on the GPU for use by another kernel, then there would be no memory transfer overhead at all. Finally, in the Monte Carlo kernel, only a small amount of data is transferred onto the device, and a small amount off; this kernel does many thousands of iterations, and the memory transfer is inconsequential. We chose what we believe to be "normal" usage cases, and we explain when data transfer assumptions can be modified; for instance, when a kernel might likely be run in between two other kernels such that the data is already present on the GPU.

Our second goal is to introduce a taxonomy that can be used for future comparisons that provides data transfer requirements while at the same time being flexible enough for the myriad of situations for which a kernel might be used. We present five broad categories for the classification of kernels, based on the memory-transfer overhead such a kernel would experience in a typical usage case: Non-Dependent (ND), Dependent-Streaming (DS), Single-Dependent-Host-to-Device (SDH2D), Single-Dependent-Device-to-Host (SDD2H), and Dual-Dependent (DD). Depending on the actual usage scenario, most kernels could fit into more than one category, but we show that when reporting performance results it is important to include results from the most likely categories, and to explain situations when the different categories would be applicable.

3.1 Summary of Novelty and Related Work

The idea for the work in this chapter came after Lee et al. published their comparison of CPU -vs- GPU runtimes, and determined that GPU speedups were generally not as impressive as much of the literature presumes [2010b]. They acknowledge that data transfer requirements can further degrade GPU speedups, but they do not provide specific research related to the issue. Other researchers have explicitly stated that data transfer between devices should be kept to a minimum (e.g., [Fan et al., 2004; Cohen and Molemaker, 2009; Dotzler et al., 2010]), but our work is the first that we know of that provides a taxonomy of data transfer requirements. It is also the first that provides a tool that can be used to compare CPU and GPU performance with the inclusion of data transfer requirements. Other researchers, such as Becchi et al. [2010], have included data transfer considerations into heterogeneous schedulers, but our work broadens the data transfer space and extends the idea to provide a comprehensive treatment of the topic. Further related work for this chapter is in chapter 7.

3.2 Benchmarks

Test Setup We chose a broad range of kernels that demonstrate a variety of memory-transfer overhead scenarios, and we chose GPUs and test CPU / PCI Express systems to show the differences that communications overhead can have depending on the speed of the GPU and the PCI Express setup in the system. All kernels were written in CUDA and were run on four platforms, as shown in Table 3.1. Of note in the table is that the GTX 480 GPU provides the fastest raw computing power but has the slowest bandwidth between the CPU and GPU, and the 330M (a laptop GPU) is the slowest card but provides one of the fastest

GPU Type	Compute Capability	Cores	Memory (MB)	Clock (MHz)	Host-Dev BW (MB/s)	Dev-Host BW (MB/s)
Tesla C2050	2.0	480	3072	1150	2413.9	2359.2
GTX 480 (Fermi)	2.0	480	1024	1401	1428.0	1354.2
9800 GT	1.1	112	1024	1500	2148.8	1502.5
330 M	1.2	48	256	1265	2396.2	2064.7

Table 3.1: GPUs Tested. All GPUs are from NVIDIA and run the CUDA programming language. The 330M GPU is a laptop GPU and the others are desktop GPUs. “Host-Dev” shows transfer times from main memory to GPU memory, and “Dev-Host” shows transfer times from GPU memory to main memory.

bandwidth of any of the devices. For some of the benchmarks, the larger data sets would not fit onto the 330M, nor onto the 9800 GT. This is reflected in the figures as fewer bars; if there are only three bars in a set, the 330M did not have enough memory to run the benchmark for that data size, and if there are only two bars neither the 330M nor the 9800GT could run the benchmark for that data size.

The PCI Express Standard The PCI Express standard provides the motherboard interconnect between the CPU and the GPU, and all memory transfers between the CPU main memory and the GPU main (shared) memory flow over the PCI Express connection. Table 3.1 shows the PCI Express bandwidth for each CPU/GPU combination, and the bandwidth results were gathered using the CUDA SDK `bandwidthTest` application for each device. As we will show in the benchmark results, in many cases data transfer over this connection contributes the majority of the time to certain benchmarks executions. It is possible to asynchronously stream data over the PCI Express while data is being acted upon on the GPU, and the CUJ2K kernel we investigated demonstrates this feature. Memory transfers that take advantage of streaming must be used on page-locked CPU memory (because it is accomplished via direct memory access), and this adds to the complexity of adding the streaming function to an application. Furthermore, regardless of the amount of streaming used, most applications require that the full amount of data passes across the PCI Express interconnect during the kernel execution, and therefore the limiting factor for most applications is still this transfer.

There is one other type of memory transfer, called "zero-copy" that can be used to transfer data back and forth from the CPU to the GPU. It is a direct memory access function that also utilizes page-locked CPU memory, and the benefits of using zero-copy are that the GPU takes full control of the memory transfers (without CPU interaction), and can also utilize the CPU main memory as if it were the main memory on the GPU. Again, however, all of the data does have to eventually pass across the PCI Express in order for the GPU to manipulate it. As with streaming memory, the speedup is still limited by the PCI Express bandwidth.

It is important to note that faster GPUs are affected more by the PCI Express overhead, simply because the amount of time they spend completing kernels is proportionally smaller than for a slower GPU with a similar PCI Express bandwidth. As new GPUs provide greater compute performance, it becomes even more important to include this overhead when reporting performance comparisons.

The Kernels and Benchmark Results Table 3.2 shows the kernels we benchmarked, and the relative amount of data that each kernel transfers to the GPU and back from the GPU after the kernel completes. For example, the Sort algorithm we analyzed requires all of the data to be present on the GPU in order to run, but it does not send any data back to the CPU. The Mersenne Twister algorithm requires only an integer as input and creates pseudorandom numbers on the GPU, but a common case is to return them to the CPU after the kernel completes.

When running our benchmarks, we focused on testing a range of data sizes, culminating in finding the largest data set that would allow the application to complete on a respective GPU. Having a large data size is important, particularly for applications that are not considered to be bandwidth bound. For instance, SGEMM, a compute bound application, still experiences a 50% slowdown for fast GPUs (in this case, the Tesla C2050 and the GTX 480) with large matrix sizes because the kernels complete so quickly.

Kernel	Data to GPU	Data to CPU
Sort	Large	None
Convolution	Large	Large
SAXPY	Large	Large
SGEMM	Large	Large
FFT	Large	Medium
Search	Large	Small
SpMV	Large	Medium
Histogram	Large	Small
Mersenne Twister	None	Large
Monte Carlo	Small	Small
CUJ2K	Large	Large

Table 3.2: Benchmark Kernels. The data size sent to the GPU assumes that the data is not already present on the GPU when the kernel launches. The data size returned to the CPU assumes the data will be used next by the CPU.

1. **Sort** is a fast radix sorting method that has been shown to sort over one billion 32-bit keys on a GTX-480 GPU [Merrill and Grimshaw, 2010]. The benchmark we tested first places all of the keys into GPU main memory, runs the kernel, and then leaves the sorted array on the GPU for further processing. Figure 3.1 shows our benchmark results with these assumptions, and it is clear that the fast GPUs with a large data set are affected the most by the memory-transfer overhead. For the GTX 480 to sort 32MB of keys, 175ms was spent transferring the data to the GPU, while only 67ms was spent performing the sort; in other words, the overall time for the sort was 3.6 times longer than for the GPU-only portion. Discussions with the author of the application we benchmarked acknowledged this overhead, but also stated that it is just as likely that the sort kernel acts on data that was on the GPU for a previous kernel. While we agree with that assessment, the fact that there is a factor of over three between the two usage cases is significant. If the data had been returned to the CPU (a common usage case), the total application would have been over six times slower than the kernel alone.

2. **Convolution** is a widely used image filtering application that can be used for smoothing, edge detection, and blur, among other functions. We benchmarked the separable convolution

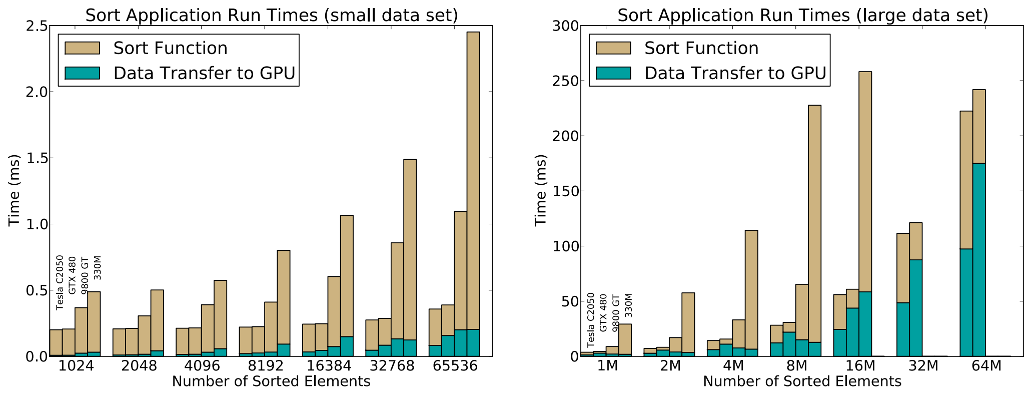


Figure 3.1: Sort benchmark. Faster GPUs are affected more by the memory transfer overhead. For instance, when sorting 64M values, the application time on the GTX 480 is 3.6x slower than the kernel itself.

---

application example from the NVIDIA CUDA SDK toolkit [Podlozhnyuk, 2007]. A typical use for an image convolution on a GPU would transfer image data to the GPU, run the kernel, and transfer the convoluted image back to the CPU, and Figure 3.2 shows the results of the benchmark. For the large data set, the application takes more time to run on the fast GTX 480 than on the slower 9800 GT because the memory-transfer bandwidth is better for the 9800 GT.

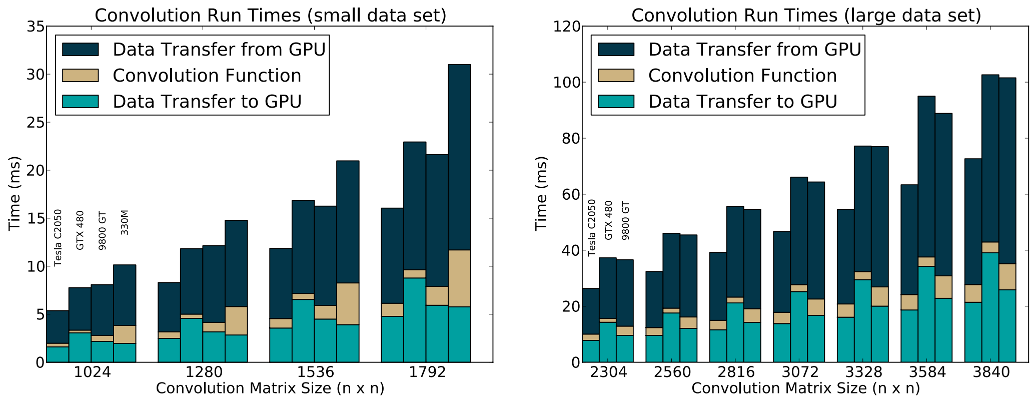


Figure 3.2: Convolution benchmark. The benchmark time is dependent on data transferred to the GPU and also on data transferred back to the CPU. Note that the slower 9800GT GPU has a faster overall run time than the much faster GTX 480 because of the slower transfer times on the GTX 480.

---

3. **SAXPY** stands for "Scalar Alpha X Plus Y" and is a function in the Basic Linear Algebra Subprograms package. It is a straightforward multiply-and-add algorithm that is pleasingly parallel. NVIDIA provides an optimized, CUDA version of SAXPY in the CUBLAS package [NVIDIA, 2011a] and this is what we benchmarked. Figure 3.3 shows the results of the benchmark, and it is obvious that the memory-transfer overhead is overwhelming to the runtime for the application. On average, the kernel plus memory-transfer times took 43x longer than the kernel processing time alone.

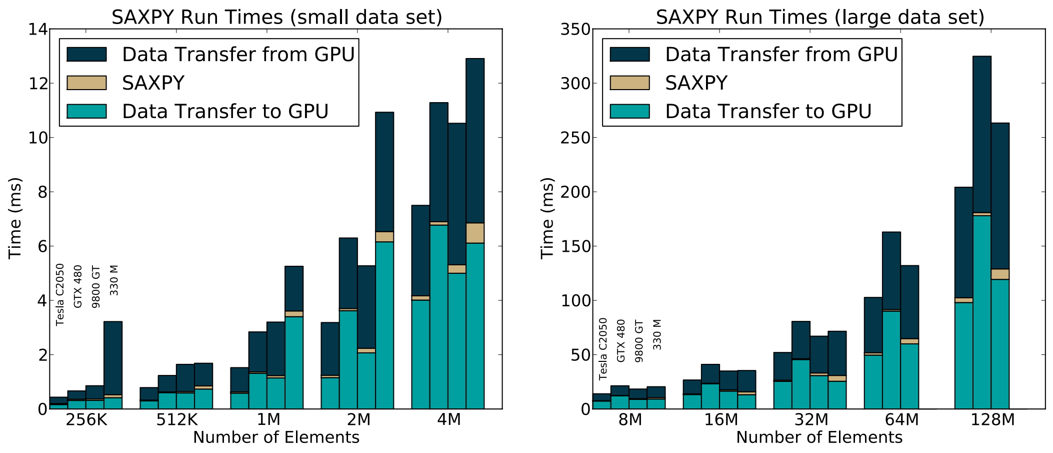


Figure 3.3: SAXPY benchmark. Because the CUBLAS version of SAXPY is well optimized and pleasingly parallel, the memory-transfer overhead comprises almost all of the application run time. Again, for large data sets the faster GTX 480 performs worse for the overall benchmark than the slower 9800 GT.

4. **SGEMM** is the Single Precision General Matrix Multiply algorithm. The CUBLAS package also includes SGEMM, and we ran our benchmark on the CUBLAS application. Like SAXPY, SGEMM is pleasingly parallel, but because it is an n^3 algorithm, it performs more calculations on the matrices than SAXPY, and Figure 3.4 shows that the memory-transfer overhead is not as overwhelming to the application as for SAXPY, especially as the data set increases. However, the run time for the Tesla C2050 is still almost twice as slow for the largest data set as it would be without the memory-transfer overhead.

5. **FFT** is the Fast Fourier Transform algorithm, which transforms signals in the time domain into the frequency domain. NVIDIA provides the CUFFT library [NVIDIA, 2011c],

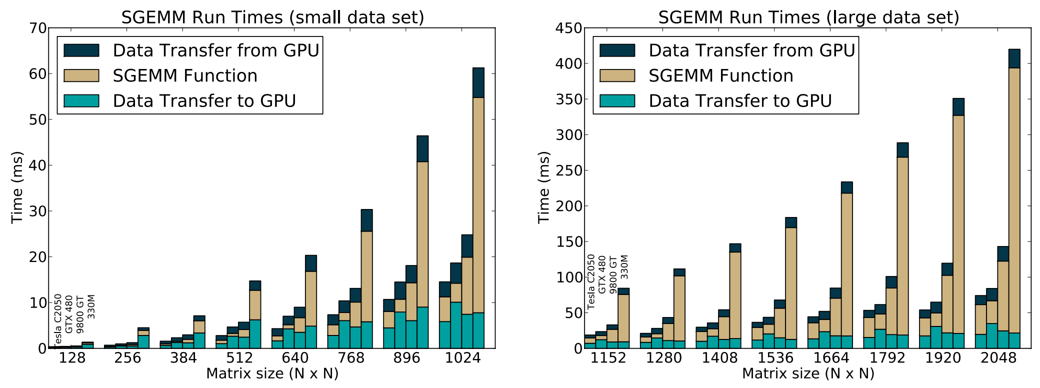


Figure 3.4: SGEMM benchmark. SGEMM is another pleasingly parallel application, but the n^3 algorithm in the kernel does spend a significant amount of time relative to the memory-transfer overhead.

which we benchmarked. Figure 3.5 shows that for large data sets, there is more than 100% overhead for data transfer on fast devices. FFT returns less data than is passed into the function, so it does take slightly less time for the transfer back from the GPU. It is still apparent that the fast GPUs are constrained significantly by the memory-transfer overhead.

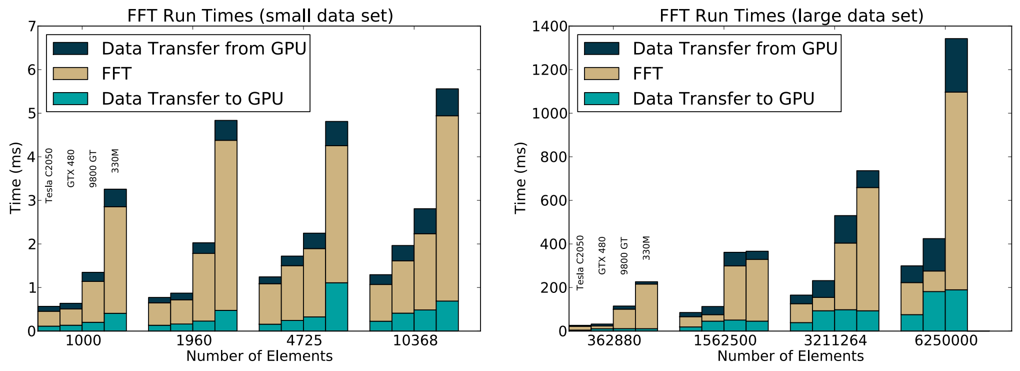


Figure 3.5: FFT benchmark. FFT produces a similar profile to SGEMM, and the fast GPUs incur a time penalty for memory-transfer of over 100% for the largest data set.

6. **Search** is a simple textual search for a short random lowercase alphabetic string in a long random string of lowercase alphabetic characters. The code we benchmarked was based on an example from an online supercomputing performance and optimization analysis tutorial [Supercomputing Blog, 2010]. For each benchmark, we measured the total time to

find 1000 different search strings in a certain length sample text. As the sample text gets larger, the likelihood for finding the search string increases, and as Figure 3.6 shows, when the size of the sample text reaches roughly two million characters, the average search time levels out. The Tesla C2050 incurs a memory-transfer penalty of $2.5x$ at this point, and the GTX 480 incurs a penalty of $5x$. The GT 9800, with CUDA compute capability 1.1*, has comparably poor performance compared to all three other GPUs because the benchmark uses atomic operations that were significantly optimized for later compute capabilities.

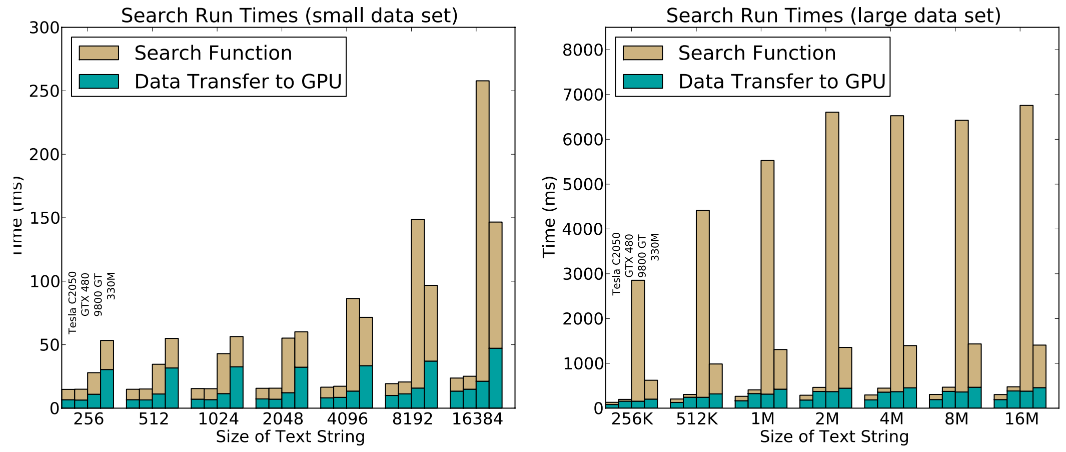


Figure 3.6: Search benchmark. The bars represent the time to search for 1000 random 4-character strings in random text string that increases in size. The text string is transferred to the GPU for each iteration. The 9800 GT's poor performance is due to the lack of optimized atomic operations in GPUs with compute capabilities less than 1.2.

7. **SpMV**, or Sparse Matrix Vector multiplication, is an important sparse linear algebra application, and it is a bandwidth-intensive operation when matrices do not fit into on-chip memory. We benchmarked an implementation described in Bell and Garland [2009] for the coordinate (or "triplet") format that has a storage size proportional to the number of non-zeros in the sparse matrix. Figure 3.7 shows the results for three different matrices (suggested by Bell and Garland [2009] and used originally for the work in Williams et al. [2009]). The application transfers much more data to the GPU than it returns, and thus

*The *compute capability* of a device denotes the CUDA features an NVIDIA GPU supports; it is akin to a version number.

the data transfer time is weighted to the amount transferred to the GPU. In the case of the largest two data sets, `mc2depi.mtx` and `webbase-1M.mtx`, the device memory-transfer bandwidth completely dictates the time for the application, and the less powerful 330M GPU completes the application faster than the GTX 480 simply because it has a higher CPU-GPU bandwidth.

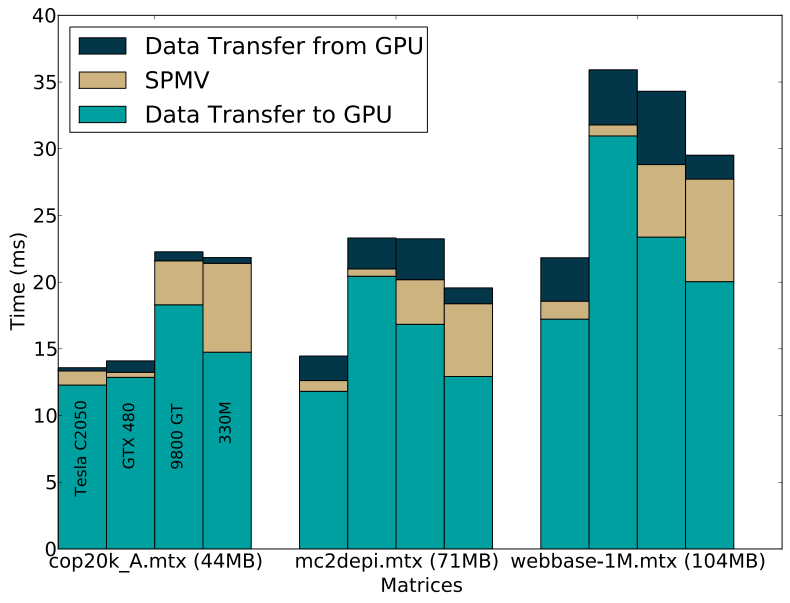


Figure 3.7: The SpMV benchmark was run on three sparse matrices stored in memory in coordinate format. The matrix size is in parentheses next to each label.

8. **Histogram** is an image processing algorithm that combines a stream of pixel light values into a series of bins that represent the distribution of light across an image. We benchmarked the CUDA SDK histogram application that computes both a 64-bin histogram and a 256-bin histogram on a set of data [Podlozhnyuk, 2011]. The histogram application sends a large amount of data to the GPU, but simply returns either a 64-byte or 256-byte array representing the bins. Therefore, as can be seen in Figure 3.8, the memory-transfer overhead for the data sent to the GPU is significant, while the memory-transfer overhead for the data sent back is not.

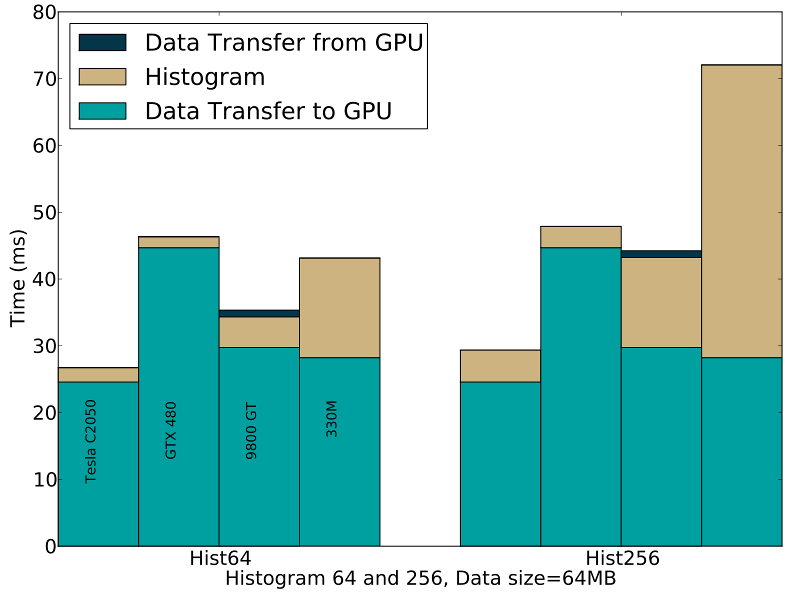


Figure 3.8: Histogram benchmark. A large amount of data is sent to the GPU, but only one small vector (64-bytes or 256-bytes) is returned to the CPU.

9. **Mersenne Twister** is an algorithm for generating pseudorandom numbers. There is no transfer of data to the GPU (except a single integer seed) and the algorithm produces the values solely on the device. Figure 3.9 shows that the only memory-transfer overhead comes from returning the results to the CPU.

10. **Monte Carlo** is an algorithm that repeatedly and randomly samples a function many times, averaging the results. We benchmarked a financial option pricing application from the CUDA SDK [Podlozhnyuk and Harris, 2008], with 2048 options, with 256K to 32M sample paths. In terms of this study, Monte Carlo is almost perfectly suited to run on a GPU, as only 32KB of data is passed to the device and 16KB is returned to the GPU, regardless of how many sample paths are computed. Figure 3.10 shows that in many cases the memory-transfer overhead is inconsequential compared to the run time of the algorithm itself.

11. **CUJ2K** is a CUDA application that converts BMP images to the JPEG 2000 format [Fürst et al., 2011]. By default it runs in asynchronous streaming mode, moving data

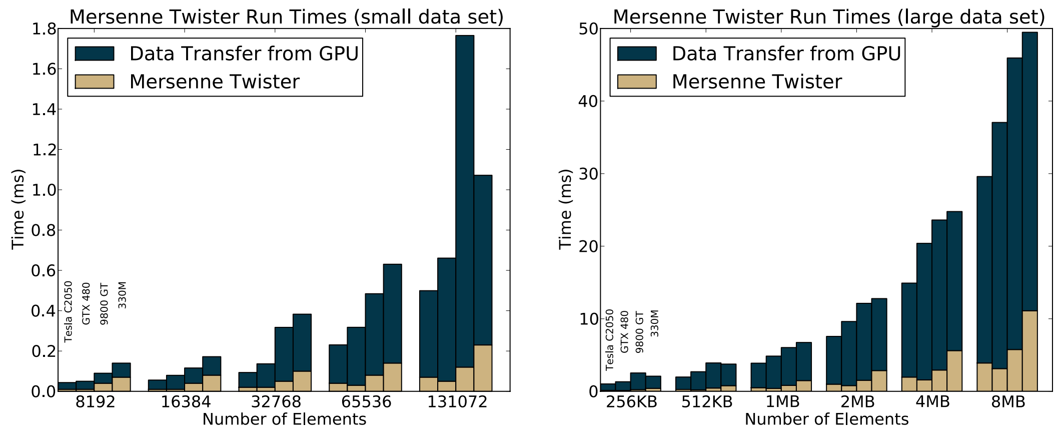


Figure 3.9: Mersenne Twister benchmark. All pseudorandom numbers are generated on the GPU, so the only memory-transfer overhead comes from transferring the results back to the CPU.

on and off the GPU at the same time it encodes data already on the device. We measured kernel times for both streaming and non-streaming conversions, as shown in Figure 3.11. For CUJ2K, asynchronous streaming improves performance when more than one image is processed by the application, and for the Tesla C2050, the speedup when converting 16 images is $5x$ over the non-streaming version. Utilizing asynchronous memory transfers is the optimal solution if data must be moved between devices, as computation and data transfer happen concurrently.

3.3 A Taxonomy for Memory Transfer Overhead

The previous section demonstrated that data transfer times are often a critical aspect of overall GPU application times. We have identified five different types of application usages that categorize the dependence on memory transfer for GPU kernels. Broadly, the categories indicate whether there is a significant dependence on transferring data to or from the GPU, which is important for both CPU/GPU comparison reporting and also for informing decisions about scheduling and other heterogeneous computing scenarios.

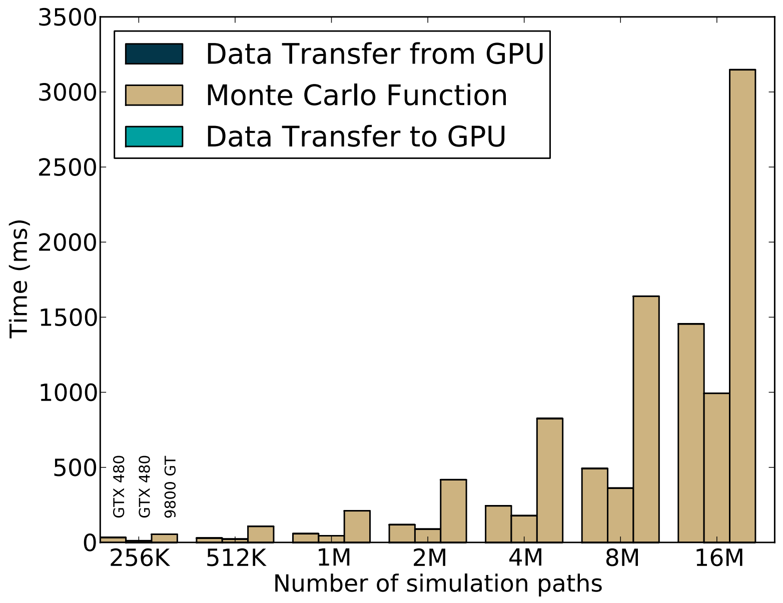


Figure 3.10: Monte Carlo benchmark. Regardless of the number of sample paths, only 32KB of data is transferred to the GPU and only 16KB is returned to the CPU. The values are insignificant compared to the overall run time of the application.

Categories of Application Kernels

GPU kernels fit into five specific categories:

1. **Non-Dependent (ND)** Those that do not depend on data transfer to or from the GPU, or the dependence is extremely small (a single value as a seed, for instance, or an integer returned as a result).

2. **Dependent-Streaming (DS)** Those that are dependent on data transfer to or from the GPU but hide this dependency with asynchronous streaming memory.

3. **Single-Dependent-Host-to-Device (SDH2D)** Those that depend on data transfer to the GPU.

4. **Single-Dependent-Device-to-Host (SDD2H)** Those that depend on data transfer from the GPU.

5. **Dual-Dependent (DD)** Those that depend on data transfer to *and* from the GPU.

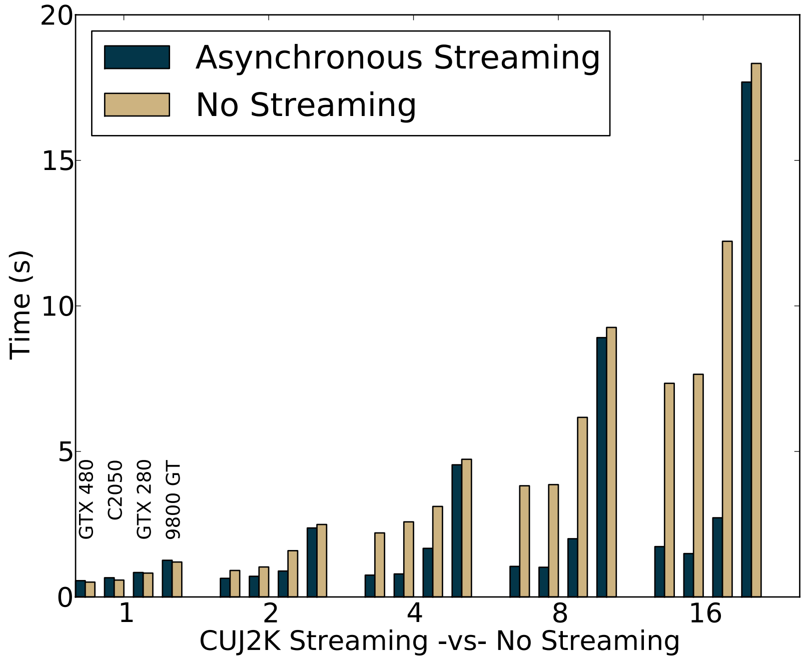


Figure 3.11: CUJ2K benchmark. The asynchronous streaming version of the application performs up to $5x$ better for faster devices. The x-axis denotes the number of BMPs that were converted to JPEG-2000 at a time. Each BMP was 2048×1361 pixels and 24-bit color. *Note: the 330M was replaced with a GTX 280 for this experiment.*

It is possible for a kernel to fit into more than one category for different usage cases; for instance a kernel that can either run independently on a set of data, or can run as a part of a sequential set of kernels on the same set of data would be in two different categories depending on the scenario. It is the responsibility of whomever is presenting the performance data to indicate which categories make the most sense for the kernel in question (as we have done in Table 3.3).

Kernels that fit into the ND category process data that is already on the GPU and either do not return the data to the CPU or return a single value or a small number of values (e.g., the integer location of search). If a sequence of kernels all act on a single set of data that is loaded onto the GPU initially and moved back from the GPU after all the kernels have run, all kernels except for the first and the last will fall into this category. The ND category provides the GPU-speedup proponents with their best results, and is indeed where most of the performance comparisons are generated.

Some applications that would not generally fit into the ND category can show similar results if they are able to take advantage of streaming memory to asynchronously transfer data to the GPU while it is simultaneously acting on part of the data. NVIDIA's CUDA has had the ability to stream data since compute capability 1.1 [NVIDIA, 2011b], and the OpenCL API supports asynchronous data transfer as well [Stone et al., 2010]. An application must specifically be written to synchronize the pipelining of such data, and there are some overheads to this synchronization.

Applications that fit into the DS category include those that implement streams to overlap data transfers with computation. Wu et al. [2009] implement a GPU version of the K-Means clustering algorithm and demonstrates the use of streaming to hide data transfer overhead. They focus on huge data sets with billions of points that do not fit onto the GPU at once.

Kernels that fit into the either the SDH2D or the SDD2H categories have similar overhead in that they must transfer data either to or from the GPU, but not both. Examples of the former would be a search algorithm that moves data onto the GPU, performs the search, and returns an integer value for the location of the search term, if found. An example of the latter would be Mersenne Twister, which produces a set of pseudorandom numbers on the GPU and returns them to the CPU. The difference between the two categories is simply at which point the memory transfer overhead is applied, at the beginning or at the end. For example, if a heterogeneous scheduler knows that there will be a memory transfer dependency at the beginning of a kernel launch, it knows that the CPU memory will be busy immediately, and it can take that into consideration for scheduling decisions.

Kernels that fit into the DD category have the most overhead to consider. Any kernel that has the sole purpose to act on data that first needs to be moved to the GPU and then moved back to the CPU main memory will be in the DD category, and this includes most of the types of kernels reported on in the literature. However, DD kernels also have the most potential to be optimized to either fit into a different category, either by streaming data, or by being incorporated into a sequence of kernels that all act on the same data.

As a rule, kernels can move between categories depending on their eventual use, as demonstrated with kernels such as sort that may work with data *in situ* as part of a larger application (or as part of a pipeline of kernels acting on a set of data), or as a standalone sorting function where the only use for the GPU is to perform the sort. GPU speedup comparisons should discuss the types of cases that would be relevant for the kernel being measured, and if there is a proposed general-use case that should be mentioned as well.

3.4 The Benchmarks Categorized

Table 3.3 shows the benchmarks from section 3.2, and the categories they fit in, based upon the benchmark results. The table also lists the secondary categories that each benchmark is likely to fall into if it was used in a more general application. For example, the search kernel we analyzed loads the entire array to be searched into GPU memory and then performs the search on the entire data set. It could be improved with a streaming memory implementation that searches the data as it arrives onto the GPU. Virtually all kernels would fit into the ND category if they were part of a sequential set of kernels and were run in between two other kernels. Similarly, if those same kernels were either the first or last of a sequential set of kernels they would either fall into the SDH2D or SDD2H categories.

Half of the applications we investigated fit into the DD category (Convolution, SAXPY, SGEMM, FFT, and SpMV). There are significant memory transfers both to the GPU before the kernels run and back to the CPU after the kernels finish. We can also envision most these applications in the ND secondary category. We placed FFT in the SDH2D secondary category because there are many signals processing applications that gather data off of a sensor in the time domain, perform FFT to move the data into the frequency domain, and then use the data directly, which could also be done on the GPU.

Application	Category	Secondary Category
Sort	SDH2D	ND
Convolution	DD	ND
SAXPY	DD	ND
SGEMM	DD	ND
FFT	DD	SDH2D
Search	SDH2D	DS
SpMV	DD	DS
Histogram	SDH2D	DS
Mersenne Twister	SDD2H	DS
Monte Carlo	ND	–
CUJ2K	DS	DD

Table 3.3: Default and secondary memory transfer categories for each benchmark. Most applications fit into the Dual-Dependent category. The secondary category lists a most-likely scenario for a more generalized inclusion of the benchmark in a real application.

It is clear from the benchmark results that the Sort, Search, and Histogram applications fit into the SDH2D as they all need data moved to the GPU in order to begin their kernels, but they do not return much data. StoreGPU [Al-Kiswany et al., 2008] fits into the SDH2D category, and the data transferred onto the GPU is many magnitudes greater than the amount that is transferred off. We have listed ND as the secondary category for Sort because there are many cases where a set of values needs to be sorted as part of a larger sequence of operations. Both Search and Histogram would benefit from streaming, and we have included DS as their secondary categories.

Mersenne Twister is the only application we benchmarked that fits into the SDD2H as a primary category. It would benefit from streaming the data back to the CPU, so we placed it into the DS secondary category. If the data was left on the GPU for further processing by other kernels, it would fit into the ND category. As seen from Figure 3.10, Monte Carlo has no significant dependencies for large data sets and fits nicely into the ND category. For some applications, such as the database applications listed in chapter 7, there is a large initial memory transfer that implies a SDD2H category, but the data then resides on the device for future kernel calculations, indicating that a long-term ND categorization would be appropriate. There are many situations in which a kernel forms a part of a larger set of calculations, and the data can reside on the GPU throughout each kernel. In this case, the kernels in the middle of this pipelined scenario would fall in the ND category. Hoberock et al. [2009] utilize a sort kernel that sorts data already present on the device and does not return it after the kernel is complete.

CUJ2K utilizes asynchronous streaming memory to partially hide data transfer between the CPU and the GPU, and therefore it has a DS classification. It can also run as a non-streaming application, so its secondary category is DD.

3.5 CPU/GPU Reporting Recommendations

We now propose a methodology for reporting on GPU performance based on our analysis of CPU/GPU memory-transfer overhead. Performance gains that include this information will provide a truer picture of not only the real speed increases that can be expected in a heterogeneous system, but will also yield more information for scheduling decisions and for more finely targeted optimizations.

Describing the "Typical Usage" of an Application: The value of a good algorithm comes when it is actually used. `Histogram` is widely used in image processing, and one common use is to take a histogram of an image and then find the minimum or maximum threshold values to use for edge detection. Therefore, a typical use would move image data onto the GPU, where a histogram kernel precedes a threshold algorithm, which could also occur on the GPU. Therefore, a typical use for `Histogram` would include moving data onto the GPU and leaving the data there for a follow-on algorithm. This is important because it demonstrates that there is a dependency for memory-transfer onto but not off of the device. If a typical usage reduces the dependency to move data from the CPU to the GPU or vice-versa, this is important to report. All typical usage cases should be included; e.g., if it is likely that a sorting algorithm will act on data that is already present on the GPU and the data will remain on the GPU, but the algorithm might also be the last kernel to run before data is transferred back to the CPU, this should be noted.

Using the Taxonomy: Our proposed taxonomy provides a broad amount of information about otherwise hidden memory-transfer dependencies of GPU kernels. `Mersenne Twister` is an excellent example: labeling it as SDD2H indicates that there is no dependence on data transferred to the GPU, but that the data will come back to the CPU for further use. Using the taxonomy is even more critical when there are different usage cases, so that someone wishing to use an algorithm can understand the performance based on how it will be used. If `Mersenne Twister` is going to return pseudorandom numbers to the CPU that will immediately be used, it would be nice to know that the algorithm is, in general, SDD2H but can also be coded as DS to indicate that it can be streamed back to the CPU as well.

Indicating the Amount of Data that will be Moved: We purposefully developed the memory-transfer overhead taxonomy to be broad in order to limit the number of categories. However, there might be cases where the amount of data transferred does not allow an algorithm to fit nicely into a specific category. A `Histogram` that includes many more than 256 bins would be an example, and it might be the case that a histogram on a small amount of data with many bins might be better described as DD instead of SDH2D. Our recommendation is to simply indicate that this is a borderline case, or to ensure that the typical use description is robust.

3.6 Heterogeneous Scheduling Ramifications

One key piece of information necessary when trying to make a decision about whether it is worthwhile to launch a GPU kernel or to perform the same function on the CPU is the projected overall time for the kernel, including all data transfers. As we have shown in this chapter, profiling the kernel on the GPU without including the data transfer times to and from the GPU overestimates the GPU speedup, and would lead to poor scheduling decisions if used. The taxonomy we propose provides a way for a scheduler to obtain the extra information about memory transfer overhead but also allows flexibility for the way a kernel will be utilized in an overall application.

We envision that a CPU/GPU scheduler will need to apply a taxonomic category to any kernel that it might schedule on a GPU. If the kernel is categorized as SDH2D and the data is only on the CPU and not on the GPU, the scheduler can calculate the added overhead time for the kernel based on the amount of data that needs to be transferred. If, on the other hand, the data is already on the GPU, the scheduler does not need to apply this overhead, and will make its decision of whether or not to launch the kernel based on this extra information.

3.7 Observations

In this chapter, we have shown that traditional reporting of CPU/GPU performance comparisons falls short because it does not adequately describe the memory-transfer overhead that applications incur in typical use cases. We benchmarked eleven diverse applications for a number of different data sizes on a range of heterogeneous platforms, and showed that the memory-transfer overhead adds a significant amount of time to virtually all applications. For some applications with large data sets, the memory-transfer overhead combined with the kernel time took longer than 50x the GPU processing time itself. However, the amount of overhead can vary drastically depending on how a GPU kernel will be used in an application, or by a scheduler. We therefore proposed a taxonomy for labeling applications to indicate these dependencies, and we made recommendations on how to utilize this taxonomy for future performance reporting.

Chapter 4

Fast Hybrid Runtime Prediction

Heterogeneous kernels may have runtimes that differ by orders of magnitude when run on different devices, but scheduling a kernel for the fastest device does not necessarily produce an overall schedule that maximizes either computational throughput or device utilization. For example, if all kernels in a workload are faster on one device, running them all on that device underutilizes the other devices, which could potentially share some of the work to increase overall throughput. Consequently, there is a need to develop effective schedulers that make scheduling decisions based on the runtime of individual kernels on the devices in a system. This chapter focuses on how to make runtime predictions that can drive such a scheduler.

A number of recent papers [Jiménez et al., 2009; Augonnet et al., 2009; Diamos and Yalamanchili, 2008; Gregg et al., 2010; Luk et al., 2009; Becchi et al., 2010] have investigated dynamic heterogeneous scheduling strategies that use kernel runtime predictions in their scheduling decision. Much of the research has focused on dynamic scheduling of heterogeneous kernels by predicting how long a kernel will run on a device and basing the scheduling data on these estimations, and the researchers have described a number of methods for making the predictions. One technique [Meng and Skadron, 2009; Hong and Kim, 2010] uses hardware performance models to predict runtimes based on how applications should behave on various platforms. However, these models are time consuming and hard to automate for new devices. Another method for making runtime predictions utilizes historical runtime data and extrapolates runtimes for queued kernels. Algorithms for making these historical data predictions have been shown to deliver precise predictions [Page, 2009; Iverson et al., 1999], but gain this accuracy with a tradeoff of time and the necessity for a large set of training data. In a dynamic scheduling situation the amount of time to make the scheduling decision can itself provide a restrictive overhead that reduces the benefits dynamic over static scheduling.

In order for a prediction model to provide sufficiently precise runtime estimates, the underlying data must show some relationship that can be exploited to make the predictions. Figure 4.1 shows nine of the ten OpenCL benchmark results run on an NVIDIA GTX 480 GPU. The graphs show kernel runtimes versus a randomly selected input parameter or a linear combination of input parameters. Five out of the ten benchmarks show a definite linear relationship. One benchmark (Reduction) shows a linear relationship but there are a few anomalous points that skew the linear regression slightly. Another benchmark (Mersenne Twister) shows a stepwise relationship, yet another (Needleman-Wunsch) shows a polynomial (x^2, indicated by a second regression line) relationship. The final two benchmarks demonstrate relationships that show some linearity but are not completely regular. Benchmarks that show linear behavior are common for GPU kernels because many serial algorithms that have this behavior can be parallelized easily and show drastic speedup when run using the GPU's SIMD engine. Benchmarks also frequently demonstrate a stepwise relationship, because the GPU's SIMD architecture requires a number of lock-step calculations, and if a set of data does not fit into the number of SIMD cores precisely, the extra processing power goes unused, leading to the same processing time for a range of data input sizes. As with any data, sometimes there is a relationship that is regular but is not reflected in the data, as the reduction example shows from Figure 4.1. As the last two graphs in Figure 4.1 show, heterogeneous applications can also demonstrate a range of other relationships between input data and runtime (including no relationship), but our experience shows that the linear is the most common.

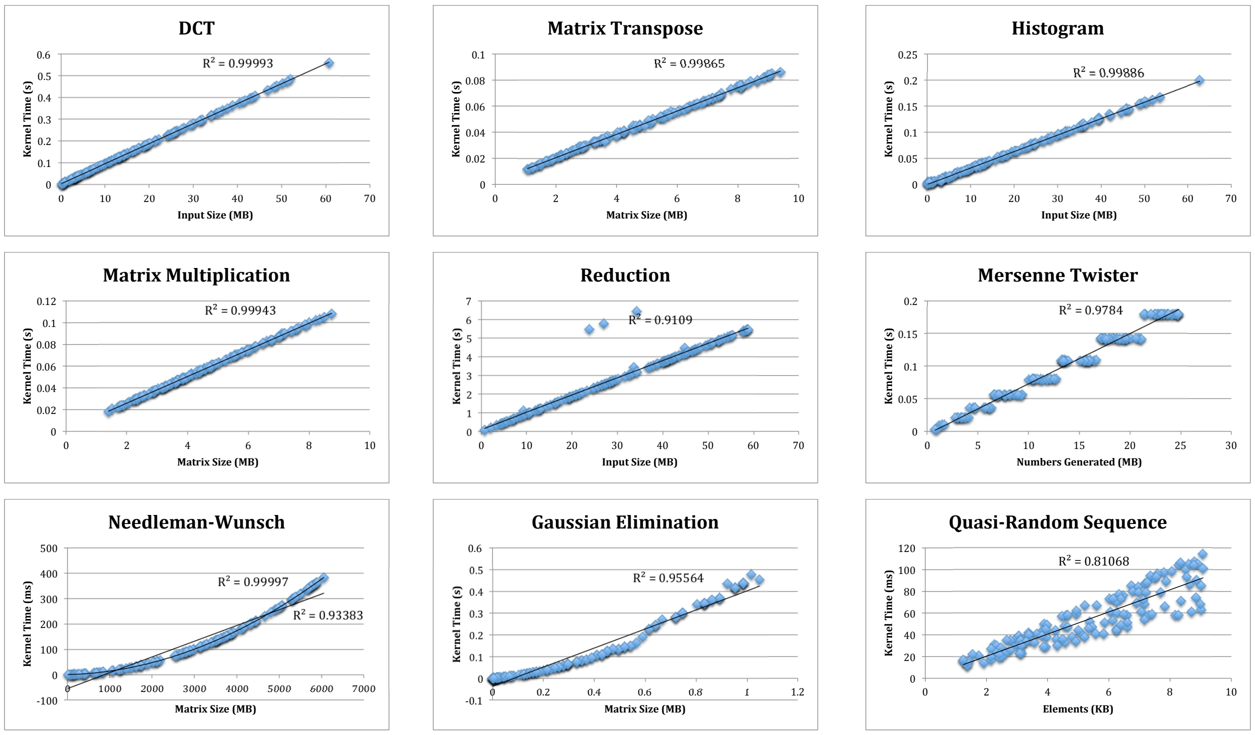


Figure 4.1: Benchmarks showing linear regression lines with R^2 values. Values close to 1 indicate excellent linear correlation.

If a kernel demonstrates a linear relationship between input characterizations and the kernel runtime, prediction for other inputs can be modeled with a simple and fast linear regression, leading to low overhead for input into a dynamic scheduling decision. Individual data about previous kernel runtimes do not need to be retained, and only aggregate values of the data (e.g., $\sum x_i$,$\sum y_i$, etc.) need to be kept in order to continue to improve the regression as new data becomes available. For kernels that are not linear, other models are available, and a model that works well across a number of relationships, given a large enough set of training data, is the k-Nearest Neighbors algorithm. k-NN does not rely on an analytical model for the relationship; instead, it finds an average of close points in the set of known values and uses that value as its prediction. k-NN is, however, computationally slower than linear regression, and using a k-NN model does necessitate retaining individual runtimes to compute nearest neighbor distances.

Utilizing historical data in the dynamic scheduling decision is sound, but we demonstrate that a hybrid model for prediction can provide better overall predictions than relying on a single model to fit different types of kernels. We also show that the precision of the prediction, while important, can be relaxed if a significant amount of time is saved by using a fast prediction algorithm. In other words, dynamic scheduling decisions that are based on runtime predictions are not adversely affected by reasonably less precise estimates, and for the most part they will determine the same schedule in most cases. We describe such a scenario in Section 4.4.

Given our assumption that a faster scheduling decision is a better one, and given that there are many GPU compute kernels that follow a linear relationship between input data and runtime, we have chosen to implement a runtime prediction model that initially assumes a linear input-to-runtime relationship and computes a fast linear regression estimation of runtimes based on random input training data. For the kernels that do not follow a linear relationship, our model falls back onto a 1-dimensional k-Nearest Neighbors (k-NN) model, which can produce a more accurate prediction but takes longer to compute and needs a larger set of training data for its accuracy. Our model can easily be expanded to include a multidimensional k-NN model if desired, but our results show that the 1-D model produces sufficiently precise runtime predictions. Our approach produces average runtime predictions that are 7% better than linear regression models alone and 5% better than k-NN models alone across a set of ten diverse benchmarks.

4.1 Summary of Novelty and Related Work

Runtime prediction itself is a relatively well-researched idea, with a number of researchers investigating k-NN prediction models (e.g., [Page, 2009; Iverson et al., 1999]). Other researchers have proposed using runtime prediction models for heterogeneous scheduling [Jiménez et al., 2009; Augonnet et al., 2009; Diamos and Yalamanchili, 2008; Becchi et al., 2010; Topcuoglu et al., 1999]. However, our work is the first to combine two runtime prediction models (linear and k-NN) into a single model that switches between the two original models based on a threshold. We demonstrate that the combined model outperforms both individual models, and also provides more precise prediction than hardware performance models, such as those described by Meng and Skadron [2009] and Hong et al. [2010]. Finally, we also show that our hybrid runtime prediction model is superior to single models for heterogeneous scheduling. Further related work for this chapter is in chapter 7.

4.2 Runtime Prediction Using Historical Data

Each time a heterogeneous kernel is run, valuable data can be collected that can be used to inform a prediction for subsequent kernel invocations. This data includes input data size, data transfer time to and from device memory, and system state such as other tasks currently running on a device. As long as the kernel has been seen at least once on a device, additional kernel runtimes can be predicted, and as more data is collected, the prediction improves. A kernel scheduler can collect the data by intercepting function calls to kernel libraries (e.g., the CUDA or OpenCL libraries) and by timing the data transfer and kernel times using simple timers or by using built-in timing functions provided by the libraries (e.g., OpenCL provides event timers for all data transfers and kernel launches). At the end of each kernel run, the running linear regression can be updated, and the next time a kernel is seen a prediction can be made with a fast calculation using the new input data size.

Figure 4.1 shows the benchmarks we tested for this research on an NVIDIA GTX 480, annotated with linear regression lines through a set of 128 random inputs. The figure also shows the associated R^2 correlation coefficient values, which shows the goodness of fit for the results; those results that have R^2 values close to 1 are considered good fits. Five of the ten benchmarks have R^2 values that are above 0.99, indicating that the linear relationship is a very good fit for the data.

For kernels that do not demonstrate a linear relationship between the input data size and the runtime, the k-NN algorithm can be used to make a runtime prediction. With a large enough historical data set, k-NN can provide an extremely precise prediction without making any assumptions about the underlying relationship between the data input and the runtime. However, k-NN takes more time than linear regression as it has to make a calculation for each data point and the input data. k-NN can also perform poorly for a small data set for which the new input data is not particularly near any of the historical data points, so it is not as useful if there are only a few previous kernel times recorded.

The database that holds the information about previous kernel data inputs and runtimes keeps the input size and kernel times for each kernel, and also holds some aggregate values in order to speed up the linear regression calculation for new kernels. The database also holds a (calculated) value for the correlation coefficient, in order to determine how well the data actually fits the linear model.

Each time a kernel finishes the values for that kernel are updated to reflect the finished kernel's runtime and data input size, and all aggregate values are updated as well. By *data input size* refers to the parameter passed to the kernel via an argument; in most cases this is a number that reflects a quantity of data, but in other cases it can reflect an iteration count (such as the number of steps in the Monte Carlo simulation).

The following two subsections describe the linear regression and k-NN algorithms in detail.

4.2.1 The Linear Regression Model

The linear regression model predicts the runtime of a kernel based on the vectors of previous input data and runtimes, (x_i, y_i). The slope, intercept, and correlation coefficient squared, (m, b, r^2), are calculated as shown in equations (4.1)–(4.3).

$$m = \frac{n \sum_i (x_i y_i) - \sum_i x_i \sum_i y_i}{\sum_i {x_i}^2 \sum_i {y_i}^2} \tag{4.1}$$

$$b = \frac{\sum_i y_i - m \sum_i x_i}{n} \tag{4.2}$$

$$r^2 = \left(\frac{n \sum_i (x_i y_i) - \sum_i x_i \sum_i y_i}{\sqrt{\left[n \sum_i {x_i}^2 - \left(\sum_i x_i \right)^2 \right] \left[n \sum_i {y_i}^2 - \left(\sum_i y_i \right)^2 \right]}} \right)^2 \tag{4.3}$$

These values can be computed online (as new values are obtained) without having to recompute the individual summations, as long as the following values are kept in the runtime database:

$$n,\ \sum_i (x_i y_i),\ \sum_i x_i,\ \sum_i y_i,\ \sum_i {x_i}^2,\ \text{and} \sum_i {y_i}^2$$

As more runtime data is collected, the correlation coefficient will either improve or degrade, indicating whether or not the data fits the linear model. In practice, values that are above 0.99 indicate an excellent fit. For small data sets (less than ten measurements), the linear model will generally provide predictions that are as good or better than the k-NN solution described below.

4.2.2 The k-Nearest Neighbors Model

If the correlation coefficient falls below 0.99 after ten or more values are in the database, the linear model will generally stop producing acceptable runtime predictions. At this point, the k-NN model should be applied to the history database for subsequent kernels. The k-NN model does necessitate using each value of the input size vector, and it therefore takes more time to produce a prediction.

The first step in the k-NN algorithm is to compute the distance from each of n input values for previous runtimes for a particular kernel stored in the history database to the input value for the kernel under consideration. With a 1-dimensional vector of size n and a new input of $n + 1$, the distance d_i is the absolute value of the difference between the two values:

$$d_i = |x_i - x_{n+1}|$$

After the distance vector $< d_0, d_1, ..., d_n >$ is created, the runtimes of the nearest k distances are averaged to establish the predicted runtime for the kernel. Determining the closest k values involves a partial sort of the distance vector, although it is possible to keep track of the closest points as the distance vector is calculated. Choosing the value for k is important, and k should grow as n grows. Iverson et al. [1999] found that $k = n^{4/5}$ produced good results for the multidimensional k-NN algorithm, but our results show that by keeping k as low as $k = n^{1/5}$ produced the best results for the benchmarks we tested.

A secondary consideration is whether or not to weigh neighbors according to how close they are to the point of interest. Giving values a weight function of $W(i) = 1/d_i$ is typical, although this tends to approximate linear regression, and we avoided this as we already know that the data is not linear. Another option is to discard any neighbors that fall outside of a given radius, even those values would be among the k closest values. This has the benefit of localizing the results, but in some cases all distances may fall outside the radius, leaving no basis for a prediction. Because we chose a smaller value of k to begin with, we do not limit the radius of results for our runtime prediction model. We will evaluate the relative speed of the k-NN algorithm compared to the linear algorithm in Section 4.3.

The following summarizes the k-NN algorithm:

1. Create a distance vector, $d =< d_0, d_1..., d_n >$ of input size values to the $(n + 1)^{th}$ input size value for the kernel runtime we want to predict.

2. Run a partial sort to find the k nearest distances, with $k = n^{1/5}$.

3. Find the average runtime of the k-nearest neighbors:

$$t_{predicted} = \frac{\sum_{k} (t(d_k))}{k}$$

4.3 Evaluation

We evaluated the linear regression and k-NN prediction models across a number of heterogeneous benchmarks and inputs in order to determine the prediction precision and overhead for each model. We found that the linear regression model was always faster than the k-NN model, and also that it provided better precision for smaller data sets. k-NN performs better for large data sets that do not show a linear relationship between the data input size and the kernel runtime.

4.3.1 Benchmarks and Workloads

For our experiments, we chose ten OpenCL benchmarks that included a wide range of compute kernels. The workloads were chosen from among the AMD and NVIDIA OpenCL SDK examples and from applications from the Rodinia [Che et al., 2009] suite that were converted to OpenCL applications. We chose typical applications that run on GPUs that show a linear relationship between input data parameters and runtime (e.g., Matrix Multiplication, Histogram, DCT), and we also chose applications that illustrate non-linear behavior, in order to demonstrate how the k-NN model is able to provide sufficiently precise predictions without knowledge of any relationship between input and runtime (e.g., Needleman-Wunsch, Mersenne Twister, and Gaussian Elimination).

We used the OpenCL event functions to directly measure kernel runtimes. We first ran each benchmark with a set of 10 random inputs but did not include the results in the historical database. We then trained the database with powers-of-two sets of inputs and after each set we tested each prediction model on the original 10 inputs. In other words, after our history database contained (2,4,8,16,32,128) data points, we ran the prediction models on each of the 10 original inputs.

4.3.2 Evaluation Platform

We ran the experiments on two GPUs and one CPU:

AMD Radeon HD 5870 GPU has 1600 Stream Processing Units (SIMD cores) running at 850MHz, and 2GB of GDDR5 memory that runs at 1.2GHz.

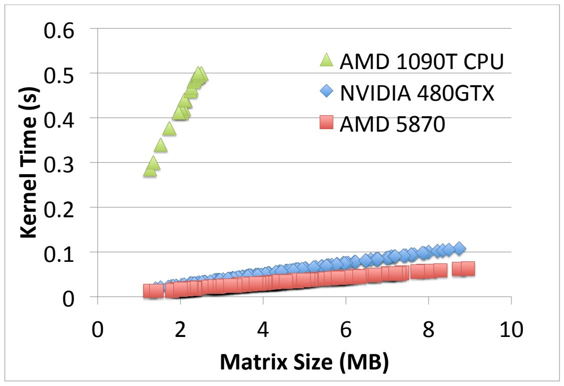
NVIDIA GeForce GTX 480 GPU has 480 CUDA SIMD Cores running at 1.4GHz, and 4GB of GDDR5 memory that runs at 1.8GHz.

AMD Phenom II 1090T CPU has 6 CPU cores running at 3.7GHz, and 4GB of DDR3 memory, 6MB of shared L3 cache, 512KB per core of L2 cache, 64KB L1 cache for instructions, and 64KB L1 cache for data.

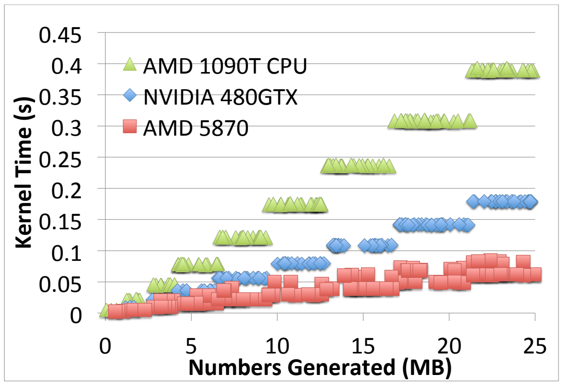
All experiments were run using the OpenCL 1.1 runtime (either AMD or NVIDIA) on the Linux Ubuntu 10.04 operating system.

4.3.3 Results

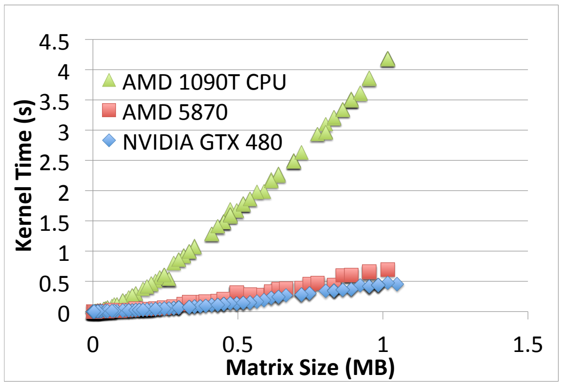
We ran each benchmark on the three different devices and in general behavior for each benchmark was similar across all devices. As examples of the similar behavior, Figure 4.2 shows the results for the Matrix Multiplication benchmark, the Mersenne Twister benchmark, and the Gaussian Elimination benchmark. All three benchmarks show similar behavior, but Mersenne Twister does not show as definitive a stepwise function on the AMD 5870 GPU as it does on the other two devices. The k-NN algorithm in our runtime prediction model accounts for this discrepancy because it does not assume an underlying pattern to the data. The R^2 linear correlation coefficient for the linear Matrix Multiply benchmarks are above 0.99 for each device, indicating a linear fit for the benchmark on all devices. The Gaussian Elimination benchmark shows that each device behaves similarly, but the data is not linear.



(a) Matrix Multiplication



(b) Mersenne Twister



(c) Gaussian Elimination

Figure 4.2: Comparison between devices. Most benchmarks show similar input-to-kernel runtime behavior across multiple devices. However, as can be seen for Mersenne Twister, the behavior is not always identical.

We attempted to fit a number curves to the Gaussian Elimination data (e.g., exponential, power), but none produced an acceptable fit.

Figures 4.3-4.6 show the results from our prediction model for benchmarks run on the NVIDIA GTX 480. The x-axis of each figure shows the training data count, which indicates how many previous runtime values are in the database. All training inputs were randomly selected, as were the inputs to the predicted values. The y-axis of each figure shows the average percent error between the predicted runtimes for ten inputs and the actual runtimes that were collected.

Figure 4.3 shows the results for the benchmarks that demonstrated a linear relationship between input data and kernel runtime. As can be seen from the graphs, the linear regression model immediately starts predicting runtimes precisely, while the k-NN model performs poorly with a small history database. As the history database grows, the k-NN model gradually approaches the precision of the linear regression model.

Figure 4.4 shows the results for the Reduction benchmark, which had some outlying data that lowers the R^2 value. The linear regression is hindered by the outlying data, but the k-NN model is not affected and continually improves as more data is collected.

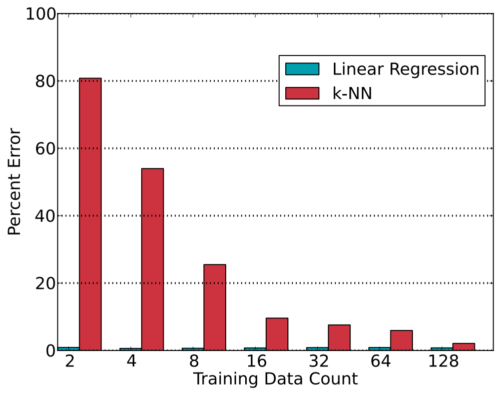
Figure 4.5 shows the results for benchmarks that demonstrate a stepwise relationship between input data and kernel runtime.

Figure 4.6 shows the results from the benchmarks that did not demonstrate a relationship that is easily quantifiable, although there is enough of a trend for k-NN to perform better as the history database grows. In the case of Gaussian Elimination, Linear Regression does not improve as more data is collected, while k-NN improves significantly. For Quasi-Random Sequence, both methods improve marginally, but the data is irregular enough that neither model can predict below 20% error on average.

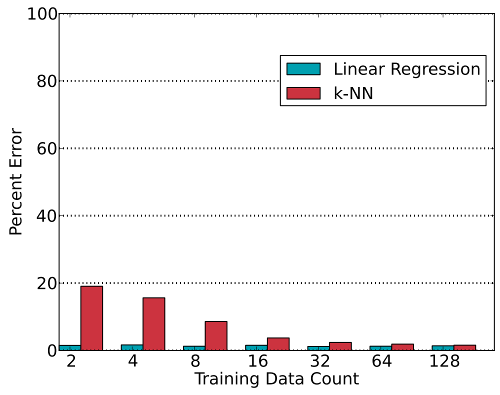
Figure 4.7 shows the increase in prediction value by using the hybrid model versus either the linear regression model or the k-NN model alone. Because the hybrid model combines the best predictions of both models for many of the benchmarks, the average prediction is better than each of the models individually.

4.4 Prediction Precision in Dynamic Schedulers

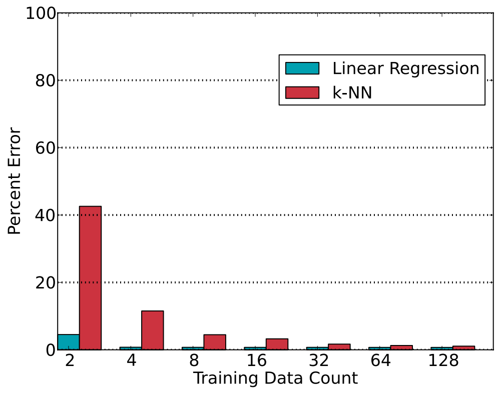
While it would be optimal to produce exact kernel runtime predictions, it is infeasible to do so, and any runtime estimation model is going to produce imprecision. As we showed in Section 4.3, runtime predictions can vary from within 5% of actual runtimes to more than 100% error. Our goal is generally to produce the best runtime prediction, but because



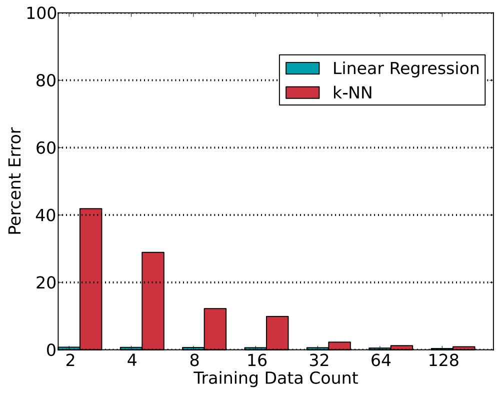
(a) DCT



(b) Matrix Transpose



(c) Matrix Multiplication



(d) Black Scholes

Figure 4.3: Prediction percent error for Linear Regression and k-NN when underlying data is linear. Both k-NN and linear regression get progressively more precise as the history database gets larger, however the linear regression model performs better when the database is small. A 1-sided, paired T-Test showed a significance level (p-value) of less than 0.05 for each set of linear-regression and k-NN averages except for the largest data set, in which case the predictions were indistinguishable. The Histogram benchmark results are not shown but demonstrate similar performance.

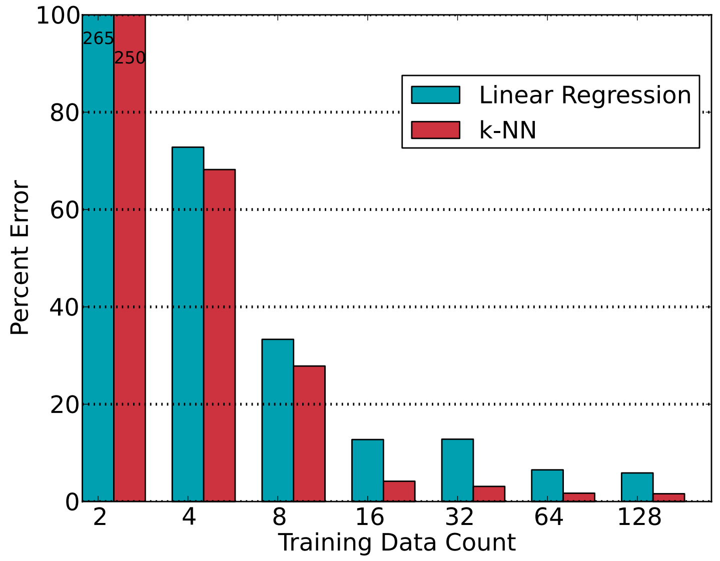


Figure 4.4: **Reduction** Prediction percent error for Linear Regression and k-NN for the Reduction benchmark. Linear regression predictions are hindered by outlying data, even though the relationship is linear. However, k-NN continually improves as the database grows, and ends up with better predictions than the linear regression. A 1-sided, paired T-Test showed a significance level (p-value) of less than 0.05 for each set of linear-regression and k-NN averages.

the linear regression algorithm can be up to 10x faster than the k-NN algorithm, it may be worthwhile to trade precision for a faster prediction in order to make the scheduling decision quicker. One consideration with heterogeneous kernel scheduling (across a number of different devices) is that runtime predictions between two devices will, in many cases, be significantly different, and therefore imprecision in the predictions will not affect the scheduling decision if that decision is based on running the kernel on the faster device. However, a number of papers [Jiménez et al., 2009; Maheswaran et al., 1999; Topcuoglu et al., 1999; Gregg et al., 2011] have discussed basing scheduling on which device will be free first or which particular kernel will finish first on a device. Both of these scheduling ideas may necessitate precise predictions in order to determine an optimal schedule.

We investigated how a scheduler based on runtime predictions would behave through a range of prediction errors. We fed the data we collected on runtimes into a simulated scheduler that uses a greedy algorithm to attempt full utilization for a system with a single CPU

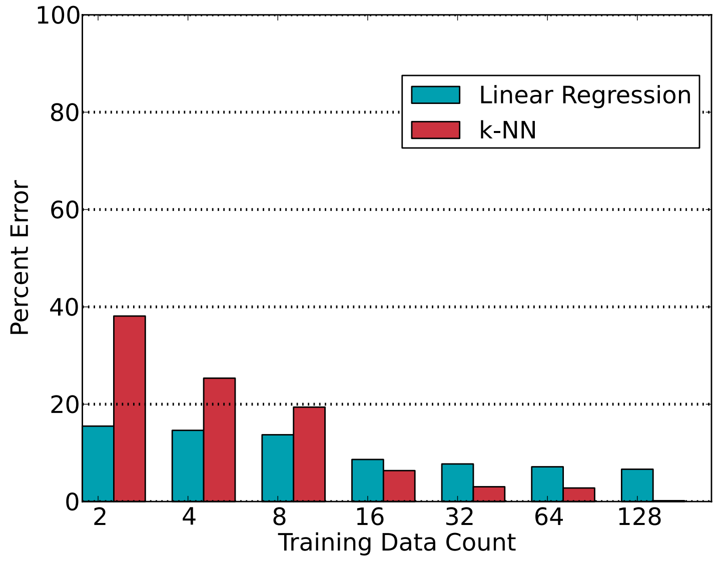
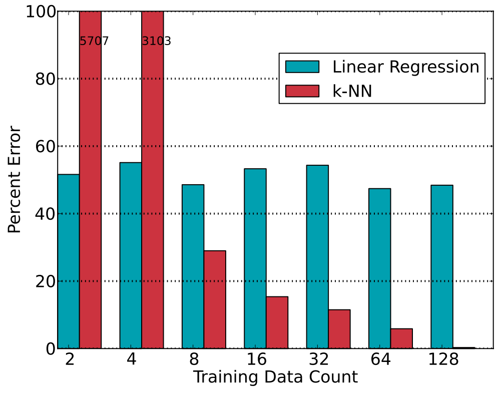
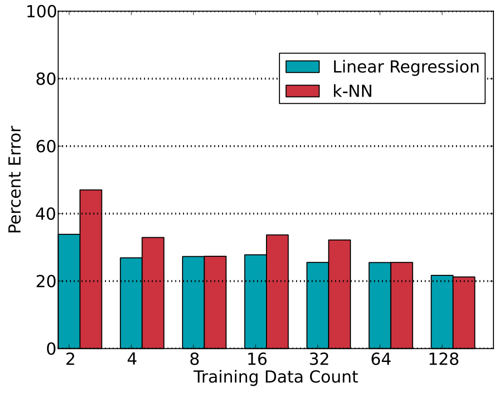


Figure 4.5: **Mersenne Twister** Prediction percent error for Linear Regression and k-NN for the Mersenne Twister benchmark. Linear regression stops improving as more data is collected. k-NN continually improves as the database grows, surpassing the linear model and becoming very precise when the history database is large. A 1-sided, paired T-Test showed a significance level (p-value) of less than 0.05 for each set of linear-regression and k-NN averages.

and a GPU. The scheduler takes a workload (in our case, ten sets of our ten benchmarks), places the kernels into a main queue and then schedules kernels onto each device based on a prediction of which device will allow the next kernel in the queue to finish first. In most cases, many kernels get scheduled on the GPU, because most of the benchmarks are between $5x$ and $10x$ faster on the GPU. However, once enough kernels get scheduled for the GPU, the scheduler will predict that it will take longer to finish those kernels than to run the next kernel in the queue on the CPU, even though it will run slower on the CPU. In this way, the scheduler keeps both the GPU and the CPU at maximum utilization, preferring the GPU in the case of a predicted tie. We ran the scheduler multiple times against different inputs with forced prediction errors of between -100% and 100% of individual benchmark runtimes. In other words, for the first experiment, we forced all predictions to be 0 seconds, and therefore the scheduler simply placed all kernels onto the GPU. In the second experiment, we forced



(a) Gaussian Elimination



(b) Quasi-Random Sequence

Figure 4.6: Prediction percent error for Linear Regression and k-NN when underlying data shows a trend that is not easily quantifiable. Linear regression starts out more precise for smaller data sets, but either stops improving or starts to get worse as more data is collected. k-NN continually improves as the database grows, surpassing the linear model and becoming very precise when the history database is large. A 1-sided, paired T-Test showed a significance level (p-value) of less than 0.05 for each set of linear-regression and k-NN averages.

all predictions to be -90% of the measured time, and we continued this pattern through 0% (perfect runtime prediction) and up to an error percent of 100%.

Figure 4.8 shows the results of one run of the scheduling experiment, but other runs with different data produced similar results. Between -20% and 50%, the scheduler produces virtually the same schedule (and in most cases the *exact* same schedule, even though the runtime predictions are incorrect. The reason this is the case is because there is a large difference between the CPU and GPU runtimes for each benchmark. When determining when a kernel will finish on a device, the relative difference in time between one device and another is generally large enough that prediction differences can be be tolerated, even if they are off by a significant percentage.

One benefit of the hybrid prediction model we are proposing is that there is flexibility in when the decision is made to switch between the two algorithms. We propose that a $0.99R^2$ correlation coefficient be used as the inflection point between the two models, but if some precision can be sacrificed in order to continue to use the faster algorithm, the value could be lowered, and the R^2 knob can therefore become a powerful one when tuning the model

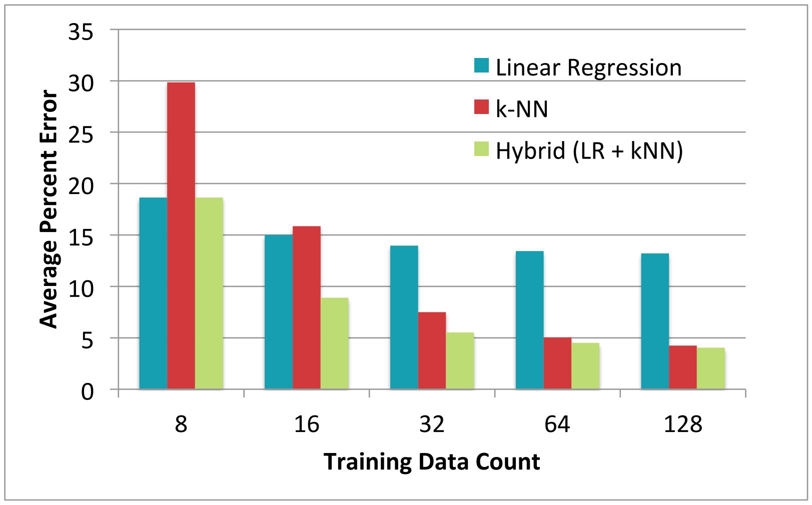


Figure 4.7: The hybrid model versus the linear regression model and the k-NN model, across all benchmarks tested. The hybrid model allows for better initial prediction with the linear model, and when the R^2 correlation coefficient for the linear regression drops below 0.99, it switches to the more precise k-NN model. The hybrid model produces better average runtime predictions than either of the other two models individually.

for a particular dynamic scheduler. Furthermore, there are other algorithms that could be substituted for either the linear model or the k-NN model, depending on whether more or less precision is needed; for example, a Lagrangian interpolating polynomial is a more complicated algorithm but could produce even better predictions for hard-to-correlate data. Another example would be to expand the k-NN model to a multidimensional model that can include more than one input parameter in its prediction. While we believe that the two-tiered model we have described is sufficient for most heterogeneous kernel prediction cases, there could also be a case made for expanding the model to include a third tier with even better precision.

Figure 4.9 shows the scheduling benefits from using the hybrid model we have described. We ran all three models with the scheduler mentioned above, and as can be seen in the figure, the hybrid model is consistently faster than either of the other models individually. The same trend can be seen as in Figure 4.7, where the linear regression model starts out well but gradually does worse compared to the k-NN model as more training data is gathered and as more benchmarks turn out to be non-linear. One of the reasons that the hybrid model does better for the overall scheduling is that it leverages the fast predictions of the linear

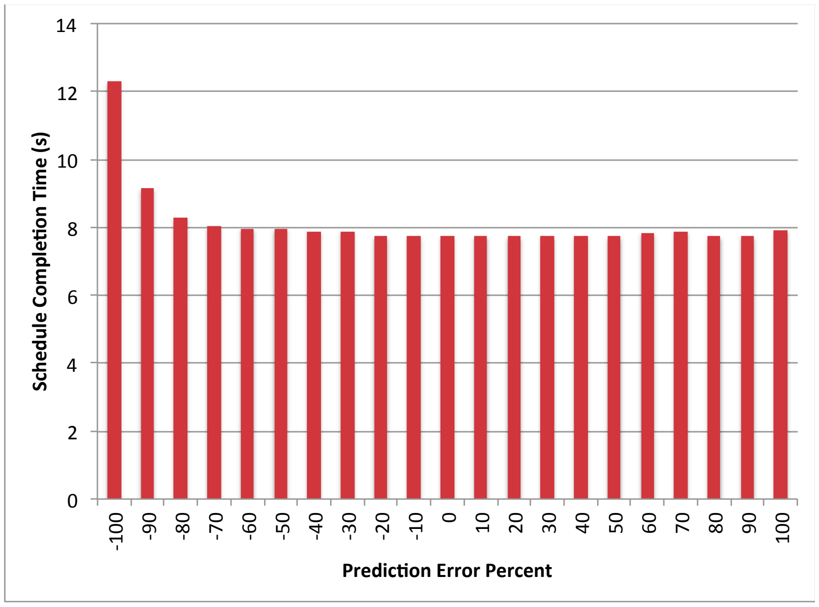


Figure 4.8: An example scheduler with forced prediction error. The far left bar shows the schedule when all kernel predictions are 0 seconds, and all kernels are run on the GPU. The middle bar at 0% shows the schedule with perfect prediction ability, and the bars to the right show over-prediction, up to 100%. The scheduler produces virtually the same schedule from -20% to 50% prediction error.

regression model for the linear benchmarks. The average time to make a scheduling decision for the linear regression model is less than one microsecond, while the k-NN algorithm takes between 10 and 15 microseconds.

4.5 Other Factors Affecting Kernel Runtimes

A number of researchers have demonstrated that kernel computation time itself is a lower bound on the overall kernel runtime, and we have shown that additional factors such as data transfer time (both from a host to a device and vice versa), kernel compilation time, and device contention can increase the amount of time a device takes to complete a kernel [Gregg and Hazelwood, 2011; Becchi et al., 2010]. Runtime predictions must take these factors into account when necessary, and in many cases they too can be quickly calculated with a minimal amount of historical data. For example, the memory transfer time between CPU memory and GPU memory (across the PCI Express bus) is linearly related to the amount of data

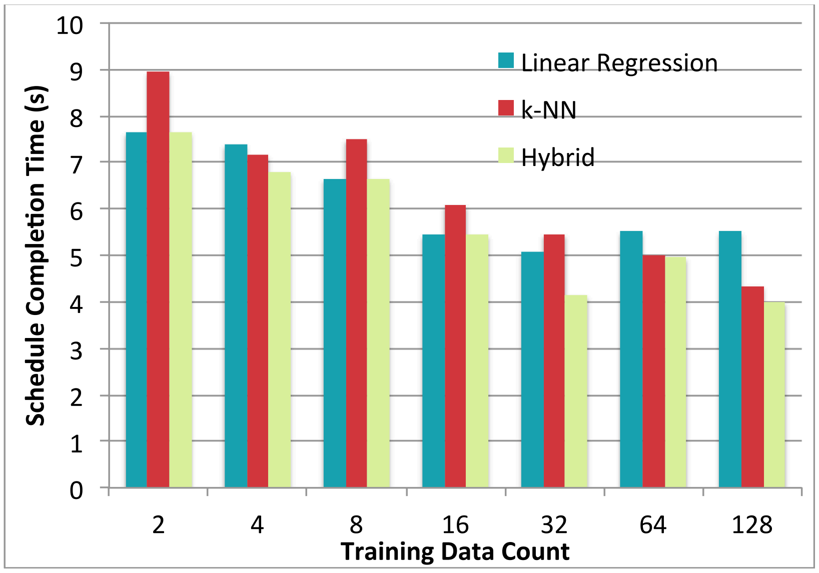


Figure 4.9: The three models providing predictions for a heterogeneous scheduler. Each model produces runtime predictions for a greedy scheduler that schedules kernels depending on the device that is predicted to finish each kernel first. The hybrid model provides better overall predictions for the scheduling decision, and the time saved with the linear regression model is significant.

transferred, and kernel compilation time is roughly constant on a given processor. A scheduler that utilizes our runtime prediction model could easily apply similar prediction models to data transfer times, kernel compilation times (if necessary), and contention information, all of which can affect kernel runtimes.

4.6 Observations

In this chapter, we described a hybrid heterogeneous kernel runtime prediction model that uses a historical database of previously run kernels. The model combines a fast linear regression algorithm with a k-Nearest Neighbors algorithm and switches between the two models if the linear regression correlation coefficient indicates that the underlying data is not linear. The linear regression algorithm works well when the database is small and particularly well for linear relationships, and the k-NN algorithm works well for hard to correlate data but is the more time consuming of the two. One clear advantage of the hybrid model is that it dynamically modifies its behavior as the historical runtime data is improved.

We evaluated the model on a number of benchmarks, and demonstrated that the hybrid model produces better runtime predictions than either the linear regression model or the k-NN model alone, and we showed that our model produces runtime predictions that are, on average within 15% of measured runtimes with eight historical data points, and can drop to under 5% error when the database is larger.

Chapter 5

Dynamic Scheduling Using Runtime and Data Transfer Prediction

In chapter 3, we described the importance of knowing the data transfer requirements for individual kernels when run on a CPU/GPU heterogeneous computer. In cases where data must be transferred between device memories, the transfer time can significantly change computational throughput and should be included in the runtime for the application. In chapter 4, we described a fast and precise model for predicting kernel runtimes. In this chapter we use the results from chapters 3 and 4 to inform a dynamic heterogeneous scheduling algorithm that schedules work between a CPU and a GPU in order to provide a high throughput for multiple applications.

As we discussed in chapter 2, in OpenCL, the task of scheduling heterogeneous kernels is left to the application itself. In general, an application chooses a device based solely on whether that device would, if available, run the kernel the fastest. Most kernels perform better on a GPU than on a CPU as they are optimized for a GPU's highly parallel architecture and GPUs typically provide higher peak throughput. Therefore, applications preferentially schedule kernels on GPUs, leading to device contention and limiting overall throughput.

Because the scheduling methodology we describe analyzes a queue of kernels that are ready to launch, some kernels later in the queue may finish prior to those that were placed into the queue earlier. Kernels can run concurrently on different devices, and therefore finish times may be out of queue order. However, the algorithm we describe does not unfairly penalize individual kernels, and we show that kernels continue to progress in the queue (i.e., they are not starved), and they will almost always finish no later than they would have finished using other scheduling methods that launch kernels in the order they were placed into the queue. We investigate the fairness of the algorithm in Section 5.4.1.

In this chapter we present an algorithm that analyzes a queue of kernels and launches them onto the devices of a heterogeneous system such that overall computational throughput is increased over a statically scheduled solution. We demonstrate that even if all kernels natively run faster on one device, there are situations where running a kernel on a slower device allows that kernel to complete before it would have if it had waited to run on the faster device. Furthermore, this solution preserves fairness for an individual kernel's placement in the queue.

We implement the scheduler on a set of twenty OpenCL benchmark applications and demonstrate the improvement of the algorithm over other scheduling decisions for a system that has a CPU and a GPU. We also show that the scheduler produces an improved schedule and a high utilization for both devices even when all kernels individually run faster on the GPU. Furthermore, we show the scheduling improvement when the scheduler takes into account data transfer time predictions verses when it ignores them.

Finally, we compare the dynamic scheduler to five other scheduling algorithms, including an oracle scheduler. We demonstrate that the dynamic scheduler that uses history data predictions and includes predicted data transfer times produces schedules that are, on average, 12% slower than an oracle scheduler.

5.1 Summary of Novelty and Related Work

Our work for this chapter focuses on heterogeneous scheduling for CPU-GPU systems. Heterogeneous scheduling in general has been heavily researched, but due to the recent advent of GPGPU computing, more researchers are targeting CPU-GPU scheduling. Although there are other heterogeneous schedulers in the literature (e.g., Jimenez et al. [2009], StarPU [Augonnet et al., 2009] and Becchi et al. [2010]), ours is the first to integrate data locality and fast runtime prediction into a task-based scheduler that targets kernels that run on either the CPU or the GPU. Furthermore, our algorithm specifically considers scheduling fairness, which is not discussed in work by others. We also compare our scheduler against a number of other algorithms, specifically HEFT [Topcuoglu et al., 1999], which is a finish-first algorithm without data transfer, and Jiménez et al. [2009], which uses a first-free algorithm to run its schedule. Our algorithm out-performs this prior work, as we demonstrate in section 5.5.3. Further related work for this chapter is in chapter 7.

5.2 Scheduling Algorithm Considerations

Scheduling computational work for heterogeneous computer systems is substantially different than scheduling for systems with homogeneous processing cores, and in this chapter we focus on two of these differences. The first difference is that a kernel can have a drastically different running time between two devices. Reports of GPU kernels running one hundred times faster than comparable CPU kernels are in the literature (for example, Che et al. [2008] and Ryoo et al. [2008]), although more recent research has shown that carefully controlled experiments demonstrate speedups that generally fall in the $2x$ to $10x$ range [Lee et al., 2010b]. Because of these runtime differences, scheduling a kernel between two devices necessitates knowing which device will run the kernel faster.

A second difference between scheduling for heterogeneous CPU/GPU systems and for homogeneous CPU systems is that GPU processors do not have the capability to time-slice workloads; i.e., kernels that are launched on a GPU run sequentially, one at a time. The latest GPUs have limited ability for multiple kernels to run in parallel, but there must be careful coordination to ensure that all kernels and their data fit onto the card, and that they do not have any dependencies. Without the ability to time-slice, a kernel launched behind other kernels must wait until all other kernels finish completely before starting its own work. In a time-sliced environment, a scheduler has the ability to interleave kernels, which enables kernels with a small amount of work to finish in a shorter time period than if they had to wait for longer kernels to finish before running. The algorithm we describe in Section 5.4 provides a similar result, because kernels farther back in the queue that will finish quickly end up on the device on where they will finish first.

Typically, when using a language framework such as OpenCL or CUDA, an application that wishes to run a kernel on a heterogeneous platform queries the system to determine which devices are available, and it preferentially chooses the device that will run the kernel the fastest. In most cases, this is a GPU, and the kernel is optimized to run on this device. Applications therefore tend to all choose the same device, and if a number of applications attempt to launch kernels concurrently, this leads to contention on a device. Furthermore, this type of scheduling ignores devices on the system that can potentially run the kernels and finish them before they would be finished if they were launched on the faster device in queue-order. We propose that instead of launching kernels to a specific device, applications should launch kernels into a queue that schedules them using an algorithm that determines the best device at a given time for each kernel. This scheduler has historical runtime information about the other kernels in the queue, and knows which, if any, kernels are currently running on each device.

Given a set of queued kernels that are not queued for a specific device, the scheduling problem becomes one of judiciously launching the kernels onto devices to maximize computational throughput while remaining fair to the queue order. Many factors can go into this scheduling decision for a given kernel, including the number of kernels ahead in the queue, the kernel or kernels that are currently running on the devices on the system and their runtimes, how much data must be transferred between device memory systems, and the relative speed of the kernels on each device in the system.

One assumption that we make in order to implement our scheduling algorithm is that kernels can be run on more than one device in a system. Our implementation utilizes OpenCL, which supports running the same kernel across both CPUs and GPUs. Kernels can be compiled for available devices prior to or at runtime, and when a kernel is launched onto a specific device, the OpenCL runtime uses the correct binary for that device. Not all frameworks support running kernels on different devices, however our scheme allows for implementations that have separate versions of a kernel for each available device (e.g., one written in CUDA for GPUs and in OpenMP for CPUs), and the runtime similarly chooses the correct binary once a device decision has been made. Indeed, if an application developer knew that a program is going to be run on a heterogeneous system, it might be appropriate to provide optimized copies of the kernels for all available computational devices. We believe that in the future more frameworks will be cross-compilable for multiple devices (as OpenCL is already), and more applications will ship with kernels that are able to run on more than one device.

In homogeneous systems, we can typically assume that a kernel's performance is independent of the processor on which it is scheduled. This assumption fails to hold in heterogeneous systems, making scheduling for heterogeneous systems fundamentally more difficult than scheduling for homogeneous systems.

5.3 Collecting and Using Historical Runtime Data

Our method for heterogeneous scheduling relies on historical data about kernel runtimes. In chapter 4 we proposed a method for collecting and amalgamating this data using a fast hybrid runtime prediction model, and we use those results to provide the history database for our proposed scheduler.

In order to train the scheduler, we run each application multiple times with different input values for each kernel. In practice, the scheduler is continually trained as new applications are added to the queue. In the results section we show the results of this training, which demonstrates that the runtime prediction model quickly stabilizes to precise values.

Application	GPU (ms)	CPU (ms)	GPU Transfer (ms)	Speedup (w/out Transfer)	Speedup (w/Transfer)
BinarySearch	69.31	48.21	672.20	0.70	0.07
BitonicSort	692.87	9991.50	538.85	14.42	8.11
BlackScholes	526.25	1613.82	207.51	3.07	2.20
DCT	102.50	212.68	726.00	2.07	0.26
DwtHaar1D	40.53	716.38	91.35	17.68	5.43
EigenValue	565.31	763.48	22.27	1.35	1.30
FastWalshTransform	172.80	1102.44	788.29	6.38	1.15
FFT	2.25	0.52	27.98	0.23	0.02
FloydWarshall	28.94	382.53	91.62	13.22	3.17
Histogram	130.46	397.91	5.85	3.05	2.92
MatrixMultiplication	42.19	7458.86	342.17	176.79	19.41
MatrixTranspose	1232.35	3747.92	2045.41	3.04	1.14
MersenneTwister	89.64	1306.31	90.23	14.57	7.26
PrefixSum	137.65	4.07	1.06	0.03	0.03
QuasiRandomSequence	10.45	368.66	300.28	35.27	1.19
RadixSort	1297.84	4360.29	1938.39	3.36	1.35
Reduction	31.50	290.24	2.09	9.21	8.64
ScanLargeArrays	16.27	138.26	507.71	8.50	0.26
SimpleConvolution	267.93	521.44	818.98	1.95	0.48
SobelFilter	5.40	6.51	3.42	1.21	0.74

Table 5.1: Dynamic Scheduling Application Benchmarks. The values shown are the actual runtimes for the input values that are used in the test scheduler. The *speedup* columns designate the GPU speedup (or slowdown) of the kernels over the CPU time, with and without data transfer times.

5.4 The Scheduling Algorithm

In this section we describe our dynamic scheduling algorithm that uses the history database described in Chapter 4. We assume that applications place kernels in a first-in-first-out queue and each kernel can run on the available devices. For clarity, we also assume that there are two devices available, a CPU and a GPU, although the algorithm could easily be extended to include an arbitrary number of devices. We also assume that most kernels will run faster on the GPU.

5.4.1 Overview of the Algorithm

In essence, the scheduler we describe implements a greedy algorithm that assigns kernels to run on a device based on a comparison between the predicted times for the kernel to finish on all available devices. Even if a kernel runs faster on a device, if there are enough kernels ahead of it in the queue for that device, it may finish faster on the slower device because that device is free.

Our scheduling algorithm is laid out as follows. We create a sub-queue for each device, and place kernels in those sub-queues from the main queue according to the following rules:

1. If the main queue contains kernels, attempt to keep all devices busy running kernels. If a sub-queue has kernels and the device for that sub-queue becomes free, run the next kernel in the sub-queue on that device.

2. If one device is busy but the other device is free and the next kernel in the main queue has not been run on that device before, run it on that device in order to build the database. This is a one-time penalty for kernels that run slowly on a device, but it is necessary to build the database. Assume that most kernels will run faster on the GPU, so if a kernel only has GPU runtime data but the GPU is free, run the kernel on the GPU.

3. If only one device is free and the next kernel in the main queue runs slower on that device, estimate how long the other device will be busy using the history database and include other kernels also scheduled for that device in its sub-queue. If the next kernel in the main queue will finish faster by being run on the slower device, run it on that device. If not, put it into the sub-queue for the busy device.

4. Continue through the main queue until both devices are busy running kernels. As a kernel finishes on a device, update the historical database with the runtime information, calculating the average runtime and the standard deviation, and repeat the algorithm from the beginning.

The scheduling algorithm described above continues to improve as more data is entered into the historical database, and each kernel is penalized at most once when it runs on a slower device in order to build the database. Runtimes and data transfer times are included in the history database. Because kernel runtimes are averaged into the previous runtime for a device, outlying points that could be caused by factors unknown to the scheduler (e.g., GPU contention due to video processing associated with the display) are smoothed out over time.

Because our scheduler places kernels on devices that are not necessary optimal for each individual kernel, we must discuss scheduling fairness. We define a schedule to be *fair* if each kernel finishes no later than it would have finished if it were allowed to execute on its preferred device. In other words, a fair schedule does not penalize a kernel even if the kernel is forced to run on its non-preferred device. Accordingly, no starvation occurs, and kernels are scheduled for a device in queue order. Specifically, kernels are always scheduled on the device for which they are predicted to finish earliest. The algorithm presented earlier generates fair schedules except in two cases: if the predicted runtimes are significantly incorrect or when a kernel is first encountered and runs on a slower device.

5.4.2 Scheduling Overhead

Although we discuss running kernels on both the CPU and the GPU as being independent from each other, this is not strictly the case. In current CPU/GPU architectures, the applications that launch kernels run themselves on the CPU, and the scheduler that runs also uses a CPU thread to coordinate the scheduling. When scheduling kernels to run on the CPU, the kernels utilize standard operating system threads, and therefore contribute to overall CPU usage. Any scheduler that launches kernels on the CPU inflicts a penalty on any applications running on the CPU, including other applications that are or will be running kernels.

Any scheduling decision incurs an overhead simply because of the time necessary to run through the scheduling algorithm. It would be imprudent if the overhead of making a dynamic scheduling decision decreased computational throughput relative to a statically scheduled solution. The algorithm we have presented is lightweight, especially compared to the application runtimes we have investigated. With the kernel database loaded into memory, our results show that the time to make a scheduling decision averages 0.7ms with a standard deviation of 0.2ms on our test platform, described in Section 5.5. GPU kernels average 273ms, making the scheduling decision average 0.3% overhead.

5.5 Experimental Results

We first describe the real workload we used to test our scheduling algorithm, and then present the experimental results compared to other scheduling algorithms. We then show simulated comparisons of all the scheduler algorithms over a randomized set of applications, in order to demonstrate the applicability across a wide range of application runtimes.

5.5.1 Workload and test environment

In order to test our algorithm, we used twenty OpenCL benchmark applications and ran the set of applications sequentially, for a total of twenty kernel launches. The applications we used in our experiments represent a number of computational algorithms that are commonly used in scientific computing. Table 5.1 shows the applications and the absolute and relative kernel run times for the data sets that we tested. As expected, most applications had kernels that ran faster on the GPU. In order to demonstrate the scheduler when some kernels were faster on the CPU, we set the data size small enough for three applications such that this was the case (Binary Search, FFT, and Prefix Sum). In Section 5.5.4 we show the result of the scheduling algorithms when all kernels are faster on the GPU, including data transfer overhead.

Our test environment was comprised of a 6-core, 3.7GHz AMD Phenom II 1090T CPU with 4GB of main memory and an AMD 5870 GPU with 2GB of memory. All tests were run on Ubuntu Linux, with the 2.6.35 kernel. In order to run and test the OpenCL applications, we wrote a Python application that places all twenty applications into a scheduling queue and schedules them according to our dynamic algorithm.

5.5.2 Training the scheduler

Figure 5.1 shows the improvement of our dynamic scheduler as the historical database improves. For the purpose of our experiments, we trained the scheduler by running the same applications in our benchmark suite, but with a random set of kernel input sizes. We also chose a sample fixed set of input sizes to test with the database after each training run, in order to normalize the results for Figure 5.1. Initially, with limited or no historical information, kernels are run on the GPU if it is free, and if the GPU is busy, a kernel is run on the CPU. Compared to a GPU-only scheduling solution, the scheduler initially performs worse as the history database is populated, but it quickly improves its performance. The database gets continually cross-trained as it collects data on all the kernels that it sees, and the scheduler benefits any time it sees a kernel more than once, regardless of the data set size. The scheduler always performs better than a CPU-only solution for this set of applications. Results in the following section reflect data taken after the scheduler has been trained for only five runs.

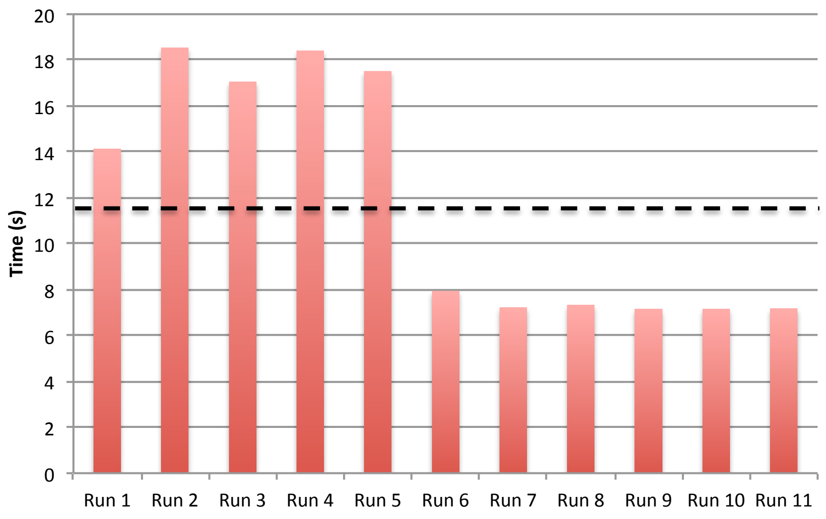


Figure 5.1: Scheduler training. The scheduler was run multiple times against the queue of twenty applications, starting with no initial historical data. The dashed line denotes the time for all applications to run on the GPU.

5.5.3 Results

Figure 5.2 shows the speedup over an oracle scheduler, of the set of twenty applications for seven different scheduling algorithms, with the oracle scheduler normalized to 1. The oracle schedule was obtained by running a brute-force exhaustive search of all possible schedule combinations. The figure shows two different dynamic scheduling results: one that includes device transfer time predictions, and one that does not. The dynamic scheduler that uses the data transfer times is only 11% slower than the oracle scheduler, and the dynamic scheduler that does not include data transfer predictions is 35% slower than the oracle, and 29% slower than the dynamic scheduler that does include data transfer predictions. The *Preferred Device* schedule schedules a kernel based on which device it runs fastest on, regardless of whether a

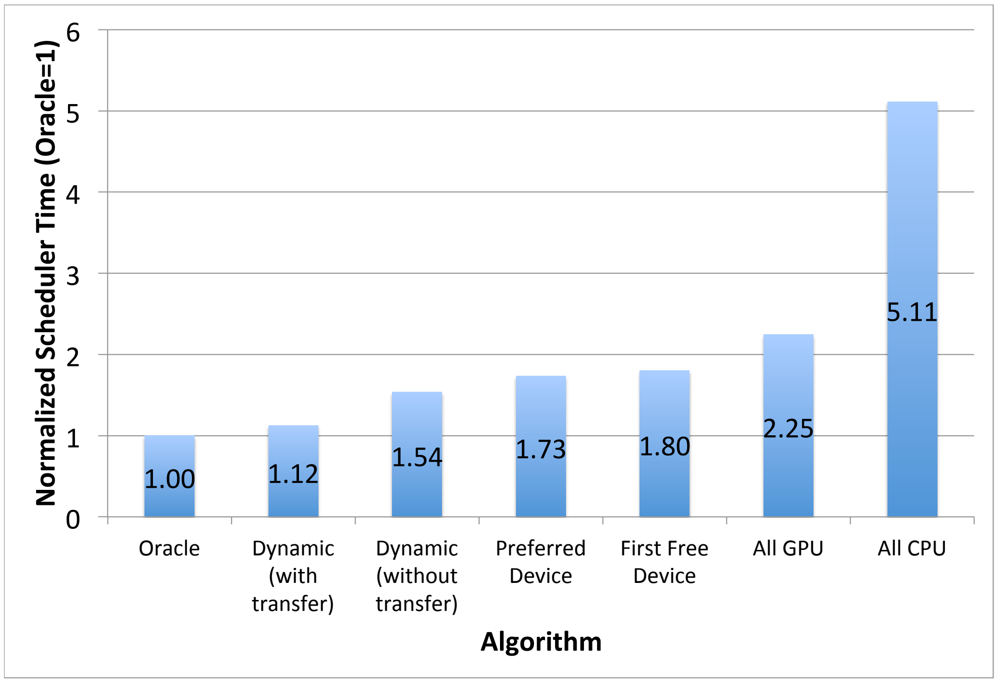


Figure 5.2: Overall results for dynamic scheduling compared against five other algorithms. The dynamic scheduler results are broken into results when the scheduler included the transfer time predictions and when it did not. *Preferred Device* uses the runtime predictions to select the device for which each kernel will run fastest, regardless of what else is running on the device. *First Free Device* places a kernel on whichever device is free, regardless of the runtime on that device. *All GPU* and *All CPU* run all kernels one device, respectively.

device is free or not. This scheduler requires runtime predictions, and produces a scheduler that is 33% slower than the best dynamic scheduler. The *First Free Device* schedule places kernels on the first device that is free, regardless of the runtime on each device. This scheduler is 38% slower than the best dynamic scheduler. Finally, the GPU only and CPU schedulers run on each kernel on their respective device, and the GPU only schedule is 50% slower than the best dynamic scheduler and the CPU only schedule is 78% slower than the best dynamic scheduler.

Figures 5.3–5.9 show the kernel schedule for each of the algorithms. Each bar represents the time that a kernel ran. Light (blue) bars indicate that a kernel ran on the GPU, and dark (red) bars indicate that the kernel ran on the CPU. The dashed line in each figure denotes the time for the dynamic scheduler that uses data transfer times for its prediction. Each figure caption reports the utilization of each device. The oracle schedule (figure 5.3) was able to obtain 100% utilization for both the CPU and GPU, and the dynamic scheduler that used

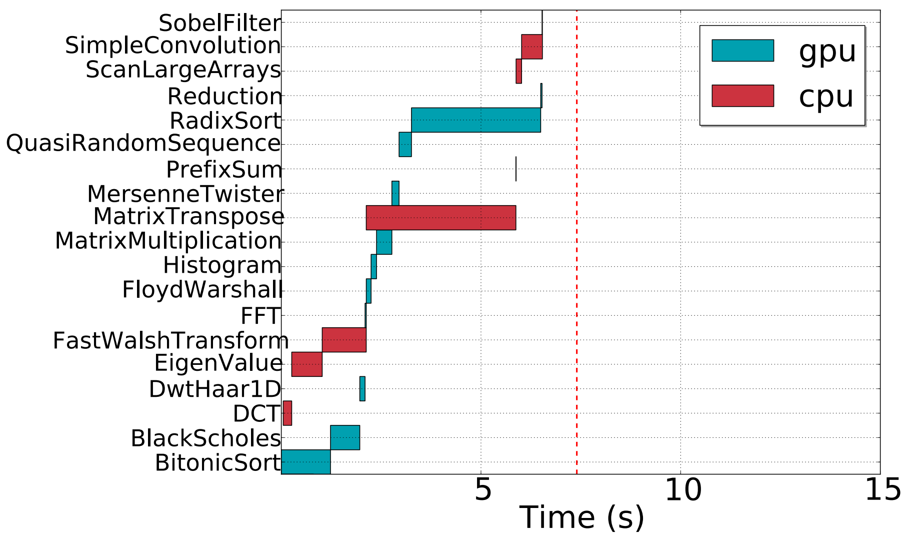


Figure 5.3: The oracle schedule. The oracle schedule was obtained by performing an exhaustive search of all possible schedule combinations. The dashed line denotes the runtime for the dynamic scheduler that includes data transfer predictions. GPU Utilization: 100%, CPU Utilization: 100%, Time: 6.5 seconds.

data transfer times was able to obtain 97% utilization on the CPU and 100% utilization on the GPU.

In order to demonstrate the utility of the dynamic scheduler across a large number of applications and runtime differences, we ran simulations of each schedule algorithm on a randomized set of kernel runtimes. Figure 5.10 shows the results of running each algorithm on one hundred different sets of twenty benchmarks. As the figure shows, the results are in line with the results from the test with real benchmarks. The dynamic scheduler that includes data transfer times is only 12% slower than the oracle scheduler, and the produces the best average scheduler compared to all other algorithms.

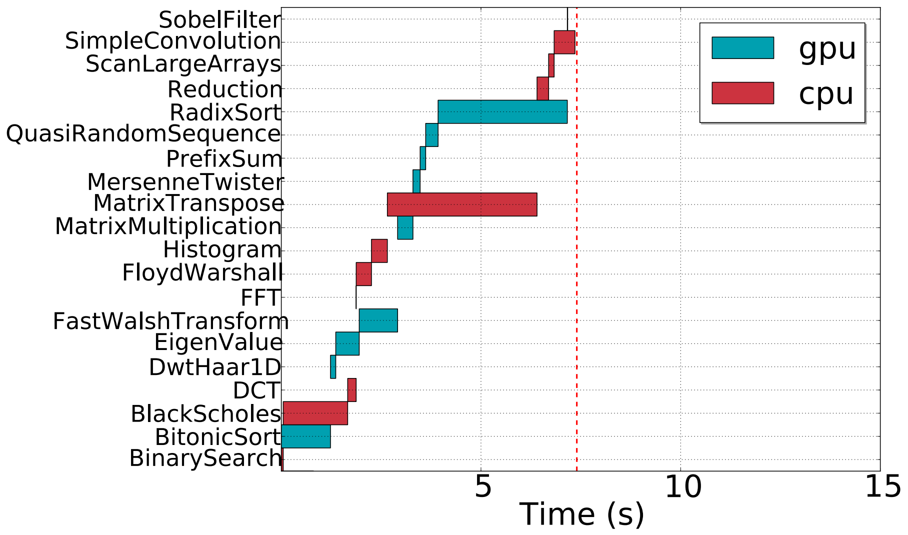


Figure 5.4: The dynamic scheduler with the inclusion of data transfer predictions. This schedule is only 11% slower than the oracle schedule. GPU Utilization: 98%, CPU Utilization: 100%, Time: 7.4 seconds.

5.5.4 Running the scheduler when all kernels are faster on the GPU

So far, we have described scheduling results from a mixed set of kernels that includes some kernels that run faster on the CPU than on the GPU. In practice, it is rare to find kernels that run faster on the CPU for other than very small data sets, as application developers generally target and optimize kernels to run on GPUs. Our dynamic scheduler is still able to produce excellent scheduling results even when all of the kernels are faster on one device, especially as kernel queue sizes increase. As an illustration, consider a queue of ten identical kernels that each run $5x$ faster on the GPU than on the CPU. As soon as the sixth kernel gets scheduled for the GPU, it is beneficial to run the seventh kernel on the CPU, even though it will take five times as long, and it will finish before it would have if it waited for the previous six kernels to finish on the GPU. Furthermore, instead of 0% utilization of the CPU, the schedule attains a very high utilization.

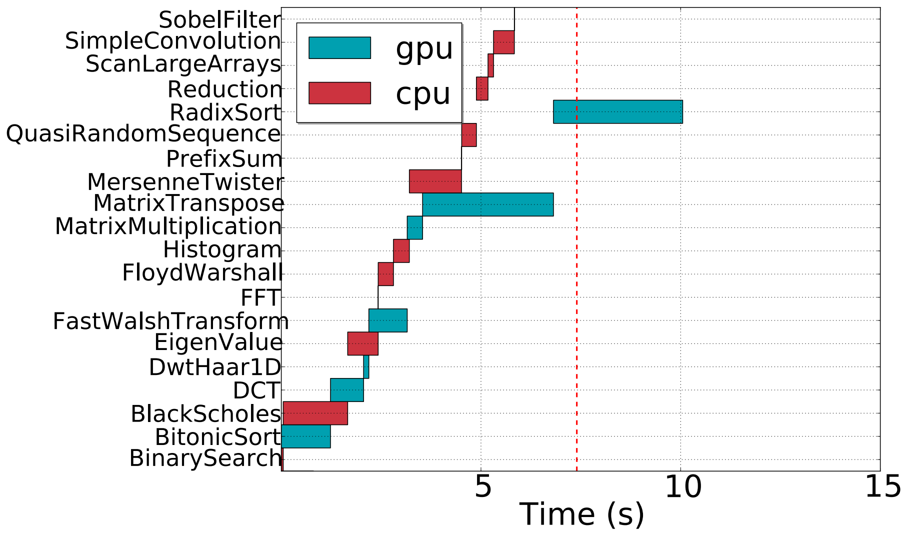


Figure 5.5: The dynamic scheduler without including data transfer predictions. This schedule is 29% slower than the dynamic schedule that includes data transfer predictions. GPU Utilization: 100%, CPU Utilization: 58%, Time: 10.1 seconds.

Figure 5.11 shows the results of running the dynamic scheduler on a set of twenty applications where all kernels run faster on the GPU. The scheduler is able to decrease the runtime for all kernels by using the CPU to run a number of kernels, and this schedule produces a 98% CPU utilization, and is only 9% slower than the oracle schedule, and the GPU-only schedule is 38% slower than the dynamic schedule.

5.6 Observations

In this chapter, we described and demonstrated a dynamic scheduling algorithm for heterogeneous computers that schedules kernels based on a historical database of runtime values. We showed that by using the historical information, a scheduler can determine how to launch kernels on heterogeneous processors that utilize all devices available and allow for a greater computational throughput than simply launching all kernels on one device, or by other static scheduling means. The resulting schedule fairly schedules kernels according to their order in the queue, and if the runtime prediction is relatively accurate, kernels will finish running

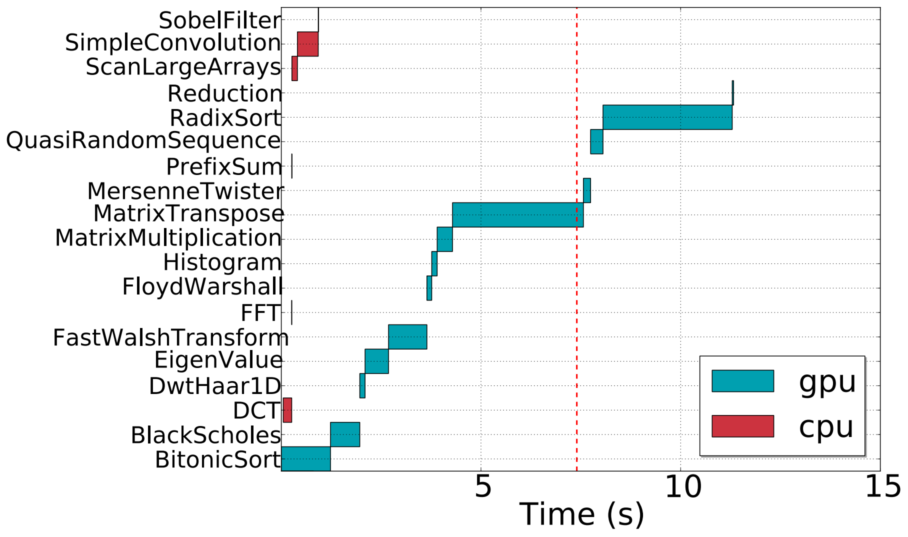


Figure 5.6: The schedule when the preferred device is used for each kernel. This schedule is 33% slower than the dynamic schedule that includes data transfer predictions. GPU Utilization: 100%, CPU Utilization: 8%, Time: 11.3 seconds.

prior to when they would have given an alternate schedule. Furthermore, we demonstrated that a scheduler that utilizes all available devices even when all kernels run faster on one particular device can still provide better overall throughput even though some kernels get scheduled for slower devices.

We demonstrated that our dynamic scheduling algorithm can increase the computational throughput for a set of twenty applications by over 1.8x over a GPU-only static scheduling solution, and we showed that the algorithm also provides device utilization that is over 90% for all devices in the system.

We compared the dynamic scheduler described in this chapter to an oracle scheduler and found that it approaches the oracle schedule time and on average provides only a 12% slowdown compared to the oracle schedule. We also compared the scheduler to five other scheduling methods, and demonstrated that it significantly outperforms them all. Finally, we showed that including data transfer time predictions in the dynamic scheduling algorithm improves the performance of the schedules produced.

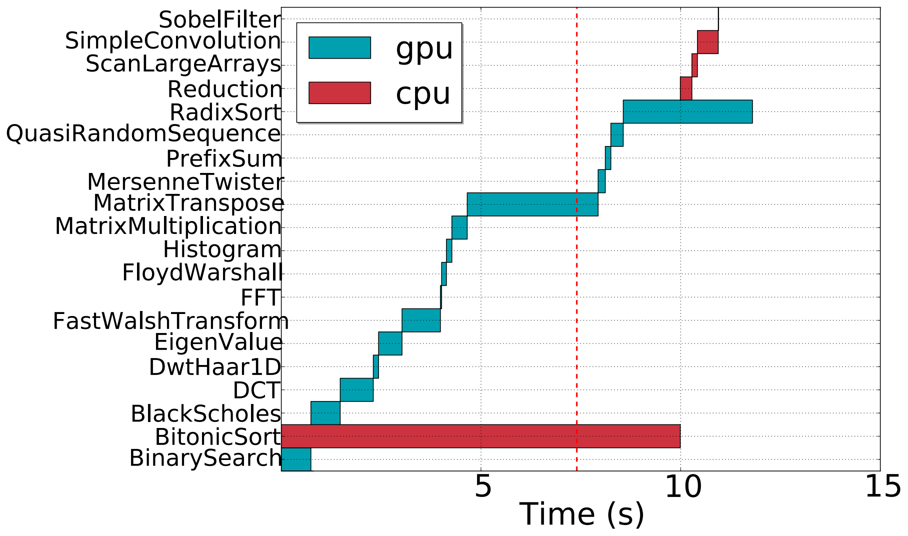


Figure 5.7: The schedule when the first free device is used for each kernel. The `BitonicSort` kernel runs much slower on the CPU than on the GPU, and this was particularly detrimental to this schedule. This schedule is 33% slower than the dynamic schedule that includes data transfer predictions. GPU Utilization: 78%, CPU Utilization: 100%, Time: 10.9 seconds.

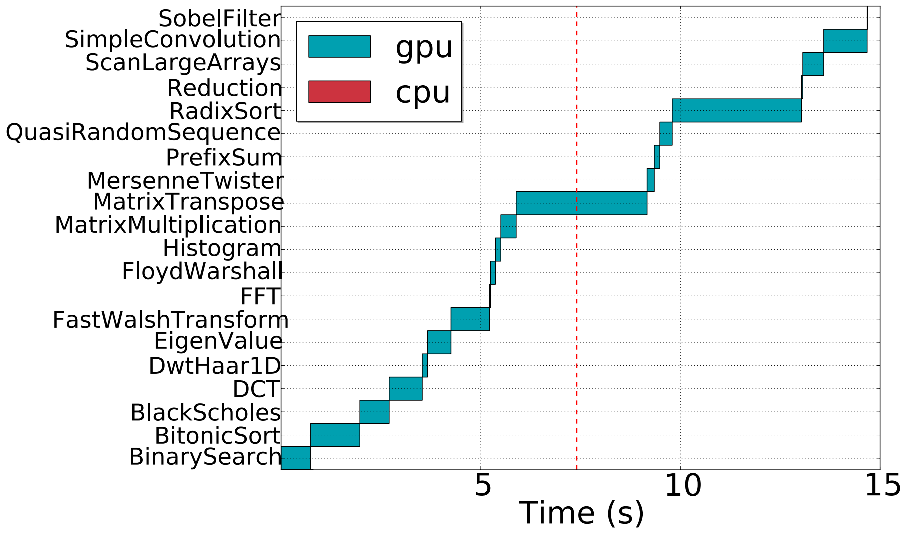


Figure 5.8: The schedule when the GPU is used for each kernel. This schedule is 50% slower than the dynamic schedule that includes data transfer predictions. GPU Utilization: 100%, CPU Utilization: 0%, Time: 14.7 seconds.

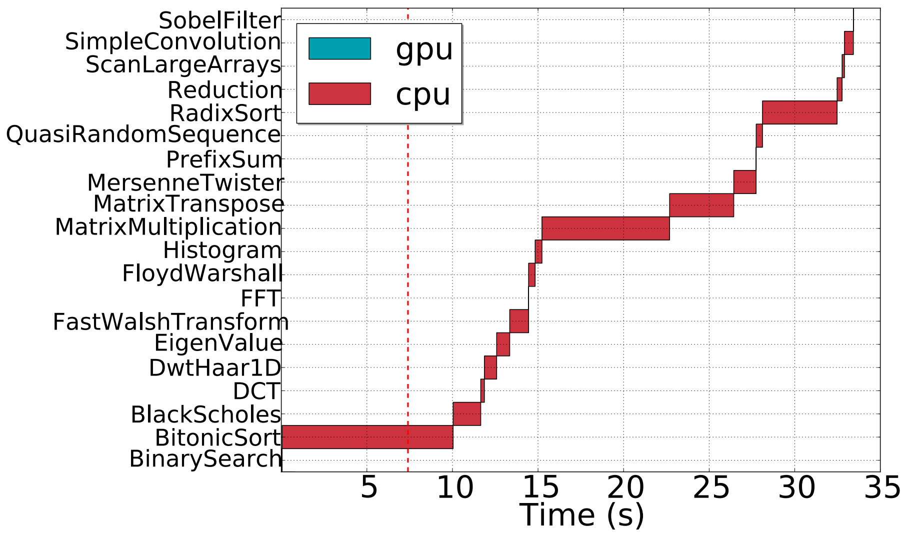


Figure 5.9: The schedule when the CPU is used for each kernel. Note that the x-axis extends to 35s on this figure as opposed to 15s on the other related figures. This schedule is 78% slower than the dynamic schedule that includes data transfer predictions. GPU Utilization: 0%, CPU Utilization: 100%, Time: 33.4 seconds.

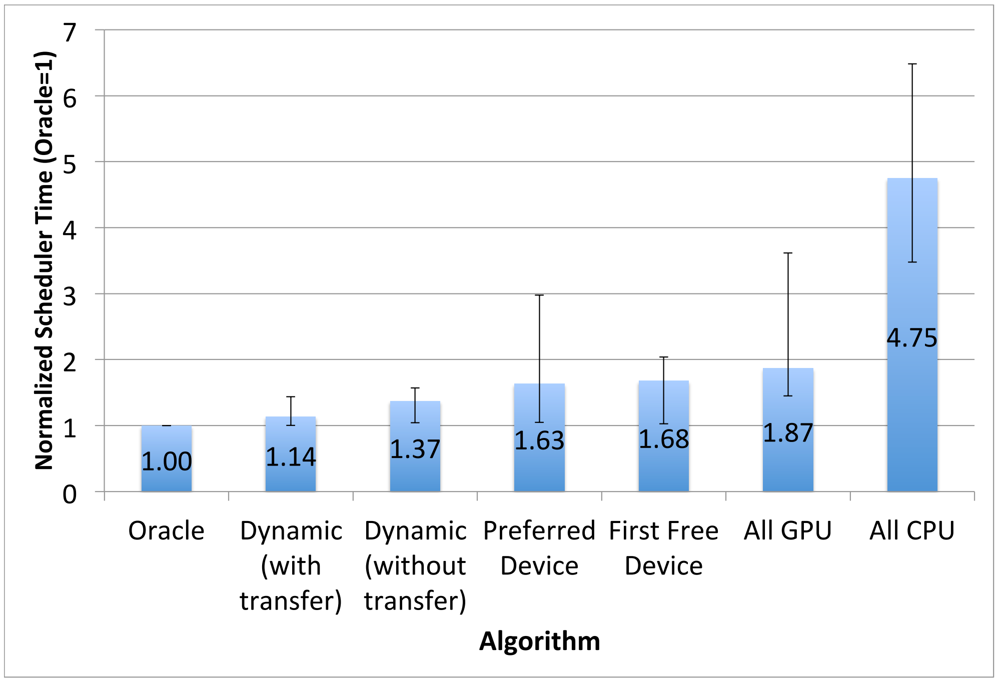
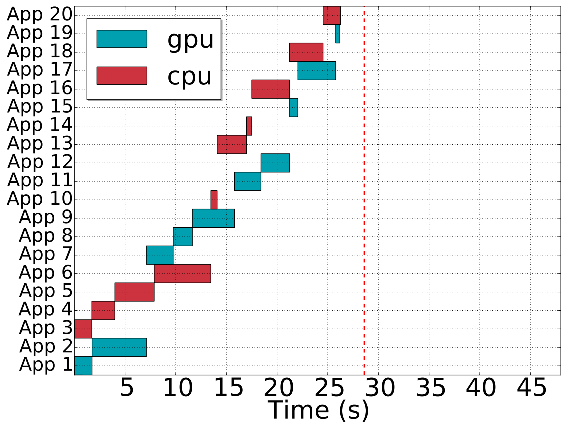
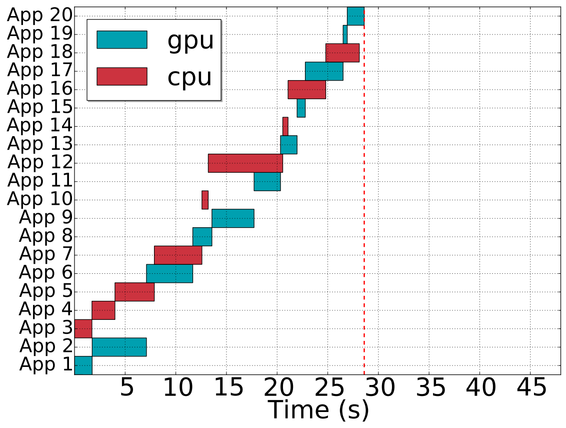


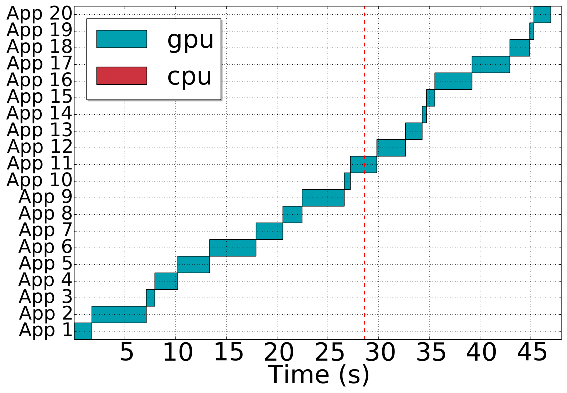
Figure 5.10: Simulated results for one hundred sets of twenty benchmarks, with randomized values for runtimes. The oracle schedule is normalized to 1. The error bars show the maximum and minimum schedule times for each algorithm. The average dynamic scheduler that uses data transfer predictions is 12% slower than the oracle scheduler.



(a) Oracle Schedule. GPU utilization: 100%, CPU utilization: 100%. Time: 26.3 seconds.



(b) Dynamic decision taking into account data transfer times. GPU utilization: 100%, GPU utilization: 98%. Time: 28.6 seconds.



(c) GPU only. CPU utilization: 0%, GPU utilization: 100%. Time: 10.1 seconds.

Figure 5.11: Execution when all kernels inherently run faster on the GPU. The dynamic scheduling finishes all applications 9% slower than the oracle scheduler but is 1.8x faster than the statically scheduled GPU-only solution. The dashed line on each figure shows the time for the dynamic scheduler.

Chapter 6

GPU Kernel Merging

In Chapters 3–5 we discussed scheduling multiple applications on different heterogeneous devices. In this chapter, we investigate how to improve multiple-kernel throughput on a single GPU. Specifically, we explore concurrent kernel execution and the utilization of GPU resources. When a kernel runs on a GPU, there are two orthogonal sources of contention: the memory system and the compute resources. When a kernel performs a large number of memory reads and writes and the memory system becomes saturated, the kernel demonstrates *bandwidth-bound* behavior. In other words, the rate at which the computation progresses is limited by the bandwidth that the memory system can provide. Conversely, when a kernel performs many computations and the compute resources limit the rate at which computation progresses, the kernel demonstrates *compute-bound* behavior.

In this chapter, we introduce *KernelMerge*, a novel prototype OpenCL kernel scheduler that we designed in order to investigate concurrent kernel execution. *KernelMerge* takes two independent OpenCL kernels and runs them concurrently on a single device. *KernelMerge* presents the OpenCL driver with a single merged kernel, with the scheduling integrated into the kernel code. Figure 6.1 illustrates the function of *KernelMerge*. In the figure, two kernels are combined into one, and the workgroups from the individual kernels are interleaved onto the device by the *KernelMerge* scheduler. *KernelMerge* also has the ability to schedule kernels such that a fixed number of workgroups from each kernel run concurrently, in essence partitioning the device and dedicating a percentage of the device to one kernel and the rest of the device to the second kernel. We describe both scheduling algorithms in section 6.2.1.

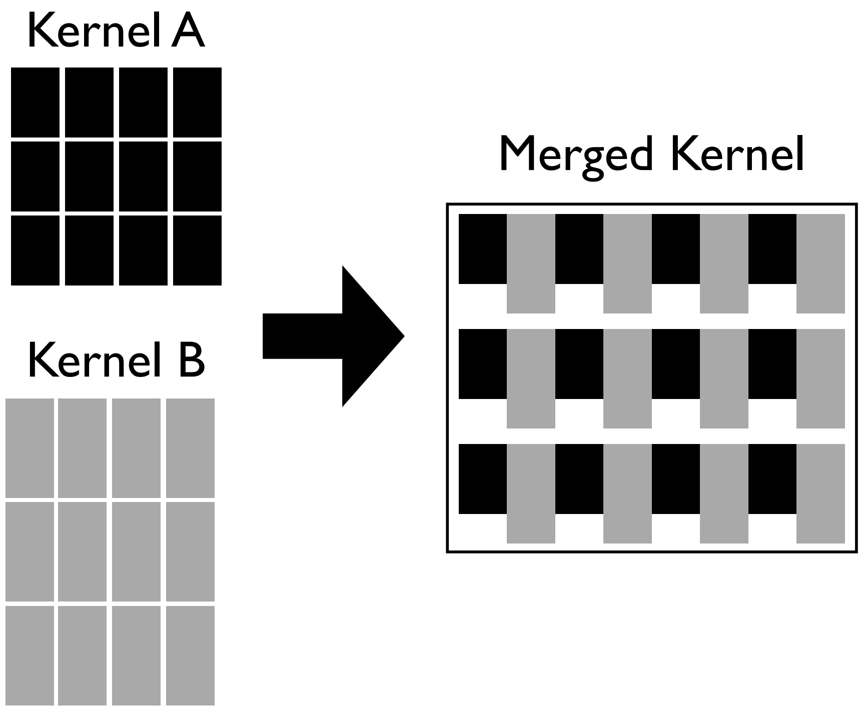


Figure 6.1: An example of the *KernelMerge* scheduler. Two independent OpenCL kernels are merged into a single kernel. In this example, workgroups from each kernel are interleaved onto a device to run concurrently. *KernelMerge* also has the ability to partition a device such that a percentage of the device is dedicated to one kernel, and the rest of the device is dedicated to the second kernel.

Even highly performance-optimized kernels may have bandwidth- or compute-bound behavior, leaving some resources underutilized. We propose that running another kernel concurrently with this first kernel can take advantage of these underutilized resources, improving overall system throughput when many kernels are available to run. Concurrent execution provides the additional benefit of exploiting the entire device, even when executing less than fully-optimized kernels. This eases the programmer burden, since optimization requires time that may be profitably reallocated.

Current GPGPU frameworks do not have the ability to run concurrent kernels with fine-grained control over resources available to each kernel, and we argue that such a feature will be increasingly necessary as the popularity of GPU computing grows. In this chapter, we demonstrate the benefits and describe *KernelMerge*. We demonstrate a case study using *KernelMerge* to find the optimal resource allocation for two kernels running simultaneously, and we show that a naïve approach to this static assignment can lead to a degradation in performance over running the kernels sequentially.

As described earlier, GPGPU code can provide tremendous speed increases over multicore CPUs for computational work. However, the nature of GPU architectural designs can lead to underutilization of resources: GPU kernels can show compute-bound or bandwidth-bound behavior, or they can be tuned to match compute and memory utilization. Application developers are cautioned to attempt to balance bandwidth- and compute-bound behavior so that a GPU kernel performs most efficiently, but this optimization can be tedious and in many cases algorithms cannot be rewritten to balance computation and memory bandwidth equally [NVIDIA, 2010; Lee et al., 2010a]. Figure 6.2 shows an example of a compute-bound kernel. To generate the figure, both the GPU memory clock and the GPU processor clock were independently adjusted for a `Merge Sort` benchmark kernel, which uses shared memory to decrease the number of global memory writes, leading to compute-bound behavior. As the figure shows, changing the memory clock frequency does not change the runtime, but changing the processor clock frequency does. In other words, the kernel is compute-bound. Figure 6.3 shows an example of a bandwidth bound kernel, `Vector Add`. In contrast to Figure 6.2, the bandwidth-bound kernel is affected by the memory clock frequency adjustments, and not the processor clock adjustments.

When a compute-bound or bandwidth-bound kernel runs alone on the GPU, it may not fully utilize the resources that the GPU has to offer. In the cases where it does not realize the ideal utilization, our goal is to change the ratio of compute instructions to memory instructions so that the device *is* fully utilized. Specifically, when a compute-bound kernel runs, the memory controllers may be idle while waiting for the computational units, and when a bandwidth-bound kernel is running, ALUs and other computational circuits may be forced

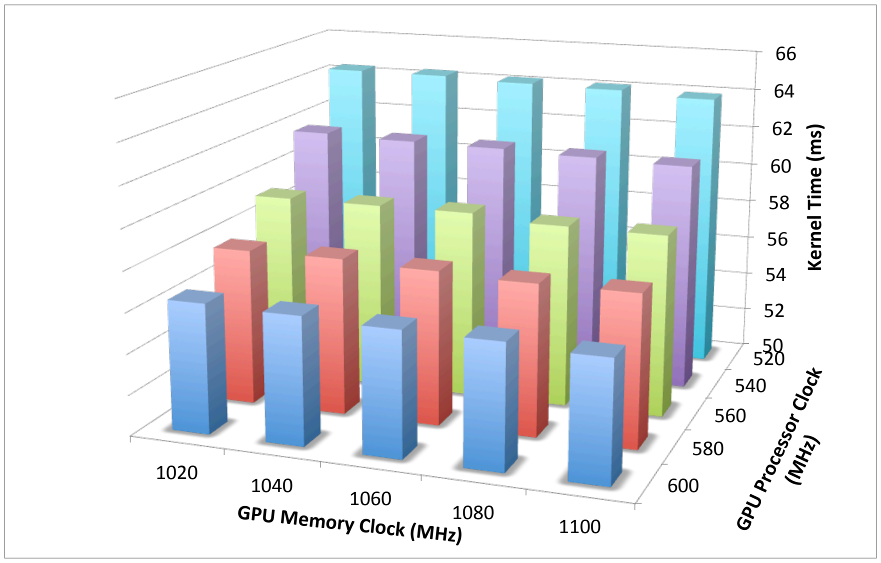


Figure 6.2: Merge Sort: a compute-bound example. The horizontal axis shows a sweep of the GPU memory clock, which does not change the kernel running time. The depth axis shows a sweep of the GPU processor clock, and as the frequency is lowered, the running time is significantly increased.

to wait for memory loads or stores. The premise for this work is that it is possible to increase utilization of the entire device by judiciously running multiple concurrent kernels that will together utilize the available resources. Using *KernelMerge* we saw up to 18% speedup for two concurrent kernels over the time to run the same kernels sequentially.

6.1 Summary of Novelty and Related Work

In this chapter, we describe *KernelMerge*, a novel OpenCL kernel scheduler that combines two kernels together and runs them concurrently. *KernelMerge* provides two scheduling algorithms (described in section 6.2.1), and provides workgroup granularity per kernel. NVIDIA's CUDA driver has limited ability to run concurrent kernels, but application developers cannot control the number of workgroups that run per kernel [NVIDIA, 2010]. As of May, 2012, OpenCL does not have the ability to run multiple kernels on a GPU. AMD has described future hardware which will include support for interrupts and exceptions, which may allow concurrent execution in a time-sliced manner. Our work-stealing algorithm (see section 6.2.1) would

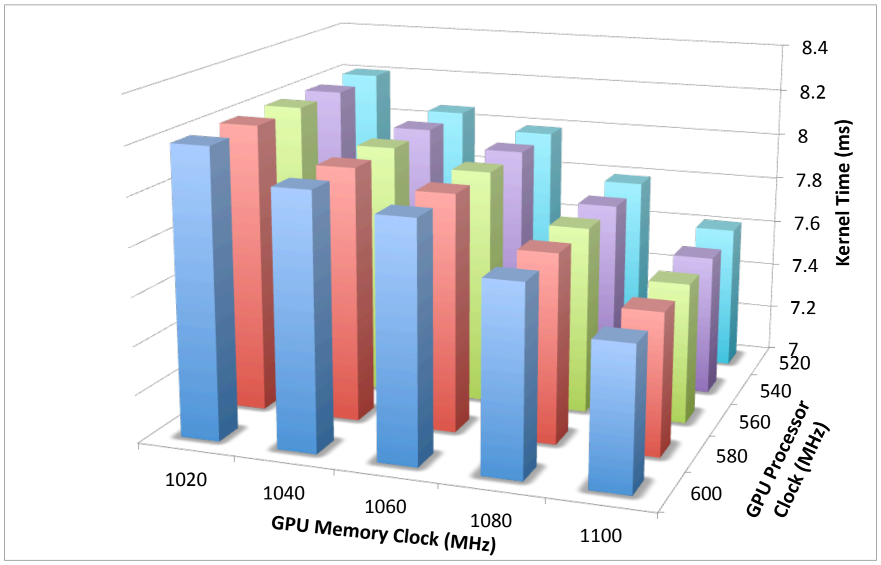


Figure 6.3: Vector Add: a memory bandwidth-bound example. The horizontal axis shows a sweep of the GPU memory clock, and as the frequency is lowered, the kernel running time increases. The depth axis shows a sweep of the GPU processor clock, which does not affect the running time.

benefit from this development. With interrupts, *KernelMerge* could provide a superior method for supplying more work to a streaming processor than what we have described in this chapter.

As we discuss in section 6.2.4, *KernelMerge* would benefit from other changes to current GPU architectures as well. NVIDIA's Kepler architecture promises a more robust ability to execute multiple concurrent kernels, but it is unclear how the available hardware will be divided in order to run multiple kernels [NVIDIA, 2012a]. If device vendors allow individual applications to declare a subset of the hardware on which to run, *KernelMerge* will be able to take advantage of that to a large extent. Further related efforts are described in more detail in Chapter 7.

6.2 KernelMerge

The goal of *KernelMerge* is to enable concurrent kernel execution on GPUs and to provide application developers and GPGPU researchers a set of knobs to change scheduler parameters.

Consequently, *KernelMerge* can be used to tune performance when running multiple kernels simultaneously (e.g., to maximize kernel throughput) and as a platform for research into concurrent kernel scheduling. For example, *KernelMerge* can set the total number of workgroups running on a GPU and can set a fixed number of workgroups per kernel in order to dedicate a percentage of resources to one or both kernels. By sweeping through these values and capturing runtimes (which *KernelMerge* provides), developers can determine the best parameters for their kernels, or researchers can see trends that may allow runtime predictions.

KernelMerge is a software scheduler, built in C++ and OpenCL for the Linux platform, that meets the need for simultaneous kernel execution on GPUs without overly burdening application developers with significant code rewrites. The C++ code *hijacks* the OpenCL runtime, providing its own implementation of many OpenCL API calls. For example, a custom implementation of `clEnqueueNDRangeKernel`, which dispatches kernels to the device, uses the current scheduling algorithm to determine when to invoke the kernel instead of the default OpenCL FIFO queue. The calling code is unchanged, using the same API call with the same arguments but gaining the new functionality. In fact most host code can take advantage of the scheduler with no changes save a single additional `#include` declaration.

KernelMerge includes additional features that are of interest to GPGPU developers and researchers. It has the ability to determine the maximum number of workgroups of two kernels that will run at once on a device, which can influence the partitioning scheme that the scheduler uses, and can also be useful when analyzing performance or algorithm design.

We envision *KernelMerge* evolving from its current form as mainly a research tool into a robust scheduler that could be implemented by an operating system for high-throughput GPGPU scheduling. We developed *KernelMerge* to allow us to run independent OpenCL kernels simultaneously on a single OpenCL-capable device. *KernelMerge* also has the ability to run single kernels by themselves, which allows measurement of scheduler overhead and also enables using the scheduler settings to investigate how a single kernel behaves.

At the highest level, *KernelMerge* combines two invocations of the OpenCL `enqueueNDRangeKernel` (one from each application wishing to run a kernel) into a single kernel launch that consists of the scheduler itself. In other words, as far as the GPU is concerned, there is a single kernel running on the device. Before launching the scheduler kernel, the scheduler passes the individual kernel parameters and buffer, workgroup, and workitem information to the device, which the scheduler uses to run the individual kernels on the device.

The scheduler creates a parametrically defined number of *scheduler workgroups*, that the device sees as a single set of workgroups for the kernel. The scheduler then dispatches a workgroup (a *kernel workgroup*) from the individual kernels to each scheduler workgroups. There are two main scheduling algorithms that we have developed: *work stealing* and *partitioning by kernel*, described in section 6.2.1. The scheduler workgroups execute their assigned kernels as if they were the assigned kernel workgroup, using a technique we call *spoofing*.

OpenCL kernels rely on a set of IDs to determine their role in a kernel algorithm. Because a kernel workgroup can get launched into any scheduler workgroup, the real global, workgroup, and workitem IDs will not correspond to the original IDs that the kernel developer had designated. Therefore, when the scheduler dispatches the kernel workgroups into the scheduler workgroups, it must *spoof* the IDs so that the kernel workgroup sees the correct values. One benefit to the spoofing is that the technique transparently allows us to concurrently schedule a kernel that uses a linear workgroup with another that uses a two-dimensional workgroup. This spoofing is responsible for a significant portion of the scheduler overhead, as discussed in Section 6.2.3. Table 6.1 shows the ID information that the scheduler spoofs.

Once the spoofing is established, the scheduler *dispatches* the workgroups which carry out the work they have been assigned. The scheduler continues to partition out the work until no work remains, at which point the scheduler (and thus the kernel that the device sees) completes.

Function	Returns
get_global_id()	The unique global work-item ID
get_global_size()	The number of global work-items
get_group_id()	The work-group ID
get_local_id()	The unique local work-item ID
get_local_size()	The number of local work-items
get_num_groups()	The number of work-groups that will execute a kernel

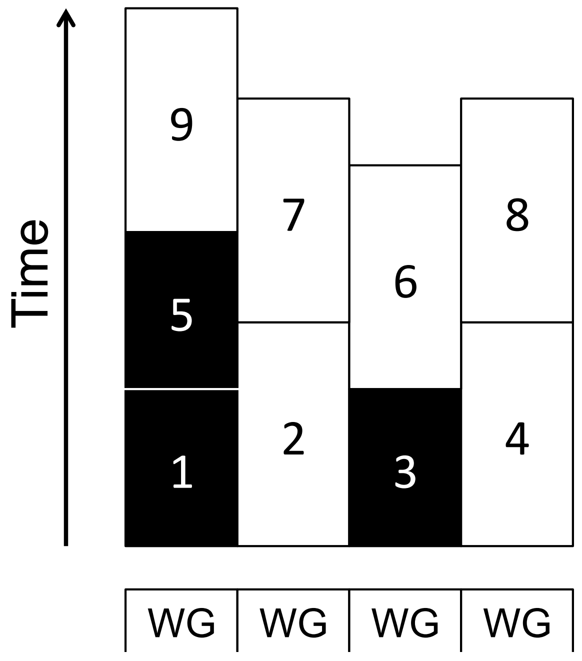
Table 6.1: Workitem and workgroup identification available to a running OpenCL kernel.

The scheduler collects runtime data for the scheduler kernel and can report on the runtime for the overall kernel. Unfortunately, because the GPU sees the scheduler as a single monolithic kernel, the scheduler cannot obtain runtimes for the individual kernels.

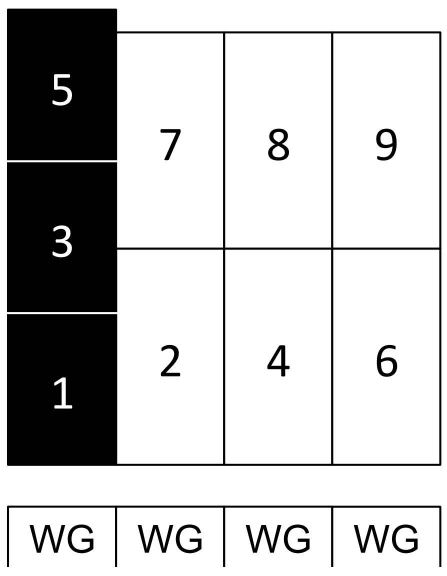
6.2.1 Scheduling Algorithms

There are two different scheduling algorithms that *KernelMerge* utilizes. The first is a round-robin work-stealing algorithm that assigns work from each kernel on a first-come, first-served basis. The second algorithm assigns a fixed percentage of workgroups to each individual kernel.

Figure 6.4 shows a graphical representation of the work-stealing algorithm on the left, and the static assignment of a fixed percentage of scheduler workgroups to each kernel on the right. For the work-stealing algorithm, Kernel workgroups are assigned alternating locations in a queue. If there are more workgroups of one kernel, the extra workgroups occupy consecutive positions at the end of the queue. Scheduler workgroups are assigned the kernel workgroups in a first-come, first-served fashion, so that at the beginning half the scheduler workgroups are running one kernel and the other half are running the other kernel. As each scheduler workgroup finishes, it is re-assigned more work from the head of the queue. With this algorithm, the allocation of resources to each kernel is not fixed, as it depends on the run time of each kernel. Once one kernel has finished running, all of the scheduler workgroups will be running the remaining kernel workgroups until it, too, completes.



(a) Work-stealing Algorithm



(b) Partitioning Algorithm

Figure 6.4: Work-stealing –vs– partitioning. The black boxes represent one kernel and the white boxes represent the other. In (b), one-fourth of the available scheduler workgroups are assigned the short-running kernel (1,3,5), and the remaining scheduler workgroups are assigned to the long-running kernel. As scheduler workgroups finish, work is replaced from their assigned kernel.

In the partitioning algorithm, when a scheduler workgroup finishes, it is only reassigned work from one kernel. This ensures that the percentage of device resources assigned to each kernel remains fixed. This algorithm runs as few as one workgroup of the first kernel while dedicating all remaining device resources to workgroups of the second kernel (and vice-versa). This ability is one of the most powerful settings available to *KernelMerge*, allowing high precision tuning of device resources assigned to each kernel. For example, if a highly bandwidth-bound kernel is run concurrently with a highly compute-bound kernel, the percentage of device resources can be adjusted so that the bandwidth-bound kernel is assigned just enough such that it is barely saturating the memory system, and the rest of the resources can be dedicated to running the compute-bound kernel, maximizing throughput for both kernels.

6.2.2 Results

Figure 6.5 shows a comparison between pairs of kernels that were run sequentially and were also run concurrently using *KernelMerge*. The figure shows that 39% of the paired kernels showed a speedup when run concurrently. However, naïvely scheduling kernels to run together can be highly detrimental; in some cases running two kernels concurrently leads to over $3x$ slowdown, because the kernels contend for the same resources. These results provide a further motivation for the use of *KernelMerge* as a research tool: being able to predict when two kernels will positively or negatively affect the concurrent runtime is non-trivial. The settings provided within *KernelMerge* have the potential to help identify general conditions that result in either beneficial or adverse concurrent kernel parings, which can in turn lead to the ability to predict runtimes and therefore make informed decisions about whether or not to run kernels concurrently. We defer the results of the partitioning algorithm to section 6.3 where we show a case study using this algorithm.

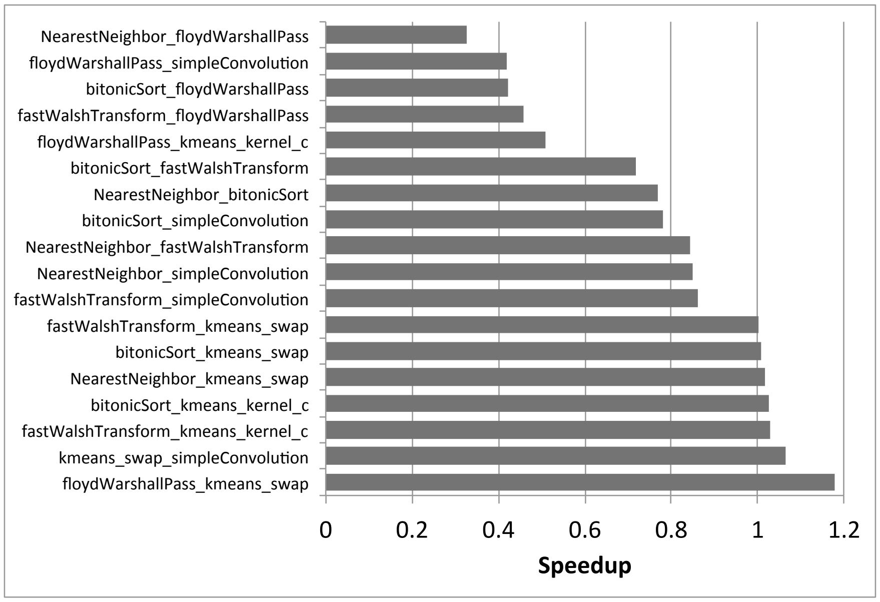


Figure 6.5: Merged runtimes compared to sequential runtimes. In this figure, pairs of OpenCL kernels were run concurrently using *KernelMerge* and compared to the time to run the pair sequentially. 39% of the pairs ran faster than they would run sequentially. However, one pair ran over $3x$ slower than the sequential runs, indicating that naïvely scheduling kernels together could lead to poor performance. Not all combinations of the seven benchmark kernels run together because some applications only contain a single kernel invocation.

6.2.3 Limitations and Runtime Overhead

As with any runtime scheduler, there are overhead costs to running kernels with *KernelMerge*. Figure 6.6 shows the overhead of running a set of kernels individually and then from within *KernelMerge*. The runtime overhead can be broken down into the following components

1. Processing a data structure in global memory to select which requested kernel workgroup to run, and to ensure that each such workgroup is run exactly once.

2. Computing the translation from the actual workitem IDs to the IDs needed for the selected workgroup (i.e., spoofing).

3. Additional control flow necessary because the compute unit ID (i.e., the ID differentiating the groups of cores) is not accessible through the API.

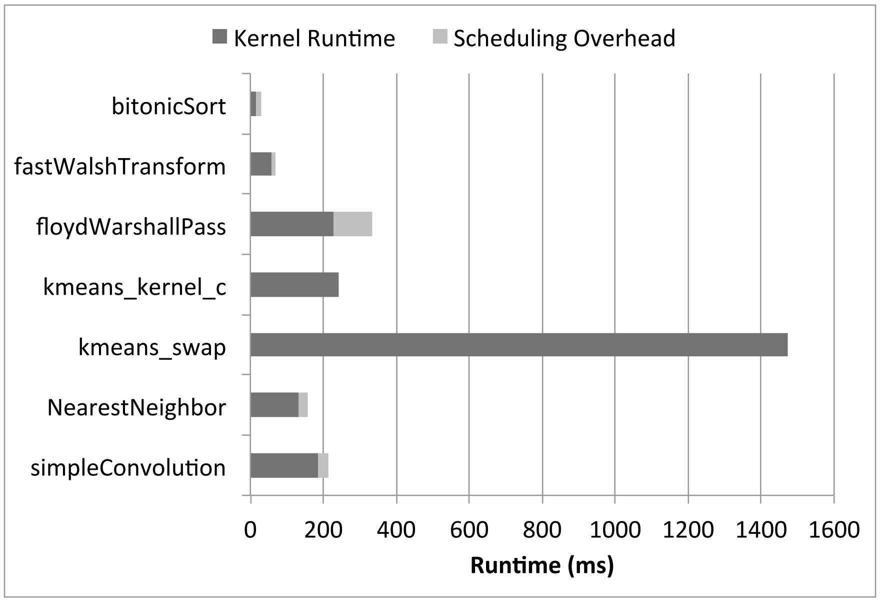


Figure 6.6: *KernelMerge* overhead. Overhead does not affect all kernels the same. Short running kernels that contain a large number of kernel workgroups are adversely affected (e.g., Bitonic Sort).

The complexity of the scheduler along with the fact that every potentially schedulable kernel is included in a single kernel combine to increase the kernel's register usage. This reduces the number of workgroups that may simultaneously occupy the device, even if only a single kernel is actually scheduled, due to limitations on the number of available registers on the device. Similarly, the number of concurrent workgroups is limited by the available shared memory; if any schedulable kernel uses shared memory, all schedulable kernels are limited by it, even if no kernel that uses shared memory is actually scheduled.

One final limitation of the *KernelMerge* approach affects the use of barriers within kernels. If two kernels that use different total numbers of workitems are scheduled, the unused workitems must be disabled. If the kernel with the smaller workgroup uses a barrier, the disabled workitems will never execute the barrier, causing the entire workgroup to deadlock.

6.2.4 Suggested GPU Architecture Changes

Many of the limitations discussed above arise because today's GPUs are built on the assumption that a single kernel runs at a time. We recommend that future GPU architectures explicitly support concurrent execution of independent kernels at workgroup granularity. We believe that implementing the scheduler in software remains the most appropriate and flexible method and that the concerns from the previous subsection can be relieved through relatively simple hardware changes. First, allowing the registers used to determine workgroup IDs to be written would eliminate the need for spoofing. Second, there are resources that need to be allocated per workgroup, such as local memory and barriers. For instance, because workgroups of different kernels may contain different numbers of workitems, the hardware must allow the scheduler to specify to the barriers how many workitems to expect. Third, exposing the compute unit ID through the API would eliminate extra control flow and would expand the space of possible scheduling algorithms. For instance, a scheduler could assign all instances of a kernel to a particular compute unit or set of compute units.

Currently, *KernelMerge* incurs significant overhead because of the necessity of working around hardware limitations to multiple kernel execution. The minimal hardware changes we suggest would enable easier creation of schedulers that implement multiple kernel execution we have presented in this chapter.

6.3 Case Study: Determining Optimal Workgroup Balance

We now show how *KernelMerge* can be used to find the optimal workgroup partitioning for concurrent kernels. Section 6.2.1 described the *KernelMerge* scheduling algorithm whereby each kernel is given a set number of workgroups out of the total workgroups available (determined by a helper application) for the kernel pair. In order to find the minimal runtime for each kernel pair, we wrote a script that sweeps through all workgroup combinations for each pair and runs the scheduler accordingly. We also ran each kernel pair sequentially and calculated the total sequential time.

Figure 6.7 shows a graph of six kernel pairs and the minimum times for each pair. For the benchmarks we used, the device (an NVIDIA GTX 460) was able to run 28 workgroups on the device at one time. The first 27 bars on the graph of each kernel pair show the combined runtime with the workgroup combinations, and the last bar on each graph shows the sequential runtime. The darker column shows the minimum runtime, and thus the optimal runtime for the pair. Many of the graphs show a *bathtub* shape, indicating that both kernels suffer significantly when their computation resources (i.e. scheduler workgroups) are restricted. In the cases that show an increasing trend towards the right, the first kernel is able to execute efficiently, even with limited computation resources. In one case (bitonicSort-kmeans-kernel-c), the sequential runtime is less than all of the scheduler runtime combinations, indicating that running the kernels concurrently is detrimental.

This case study demonstrated that *KernelMerge* can be used for determining the optimal workgroup partitioning for concurrent kernels, and it also demonstrates that naïvely scheduling the kernels together can produce undesirable results.

Figure 6.8 compares the results of the work-stealing algorithm to the sweet spots found by the partitioning algorithm. As the figure shows, the work-stealing algorithm approaches

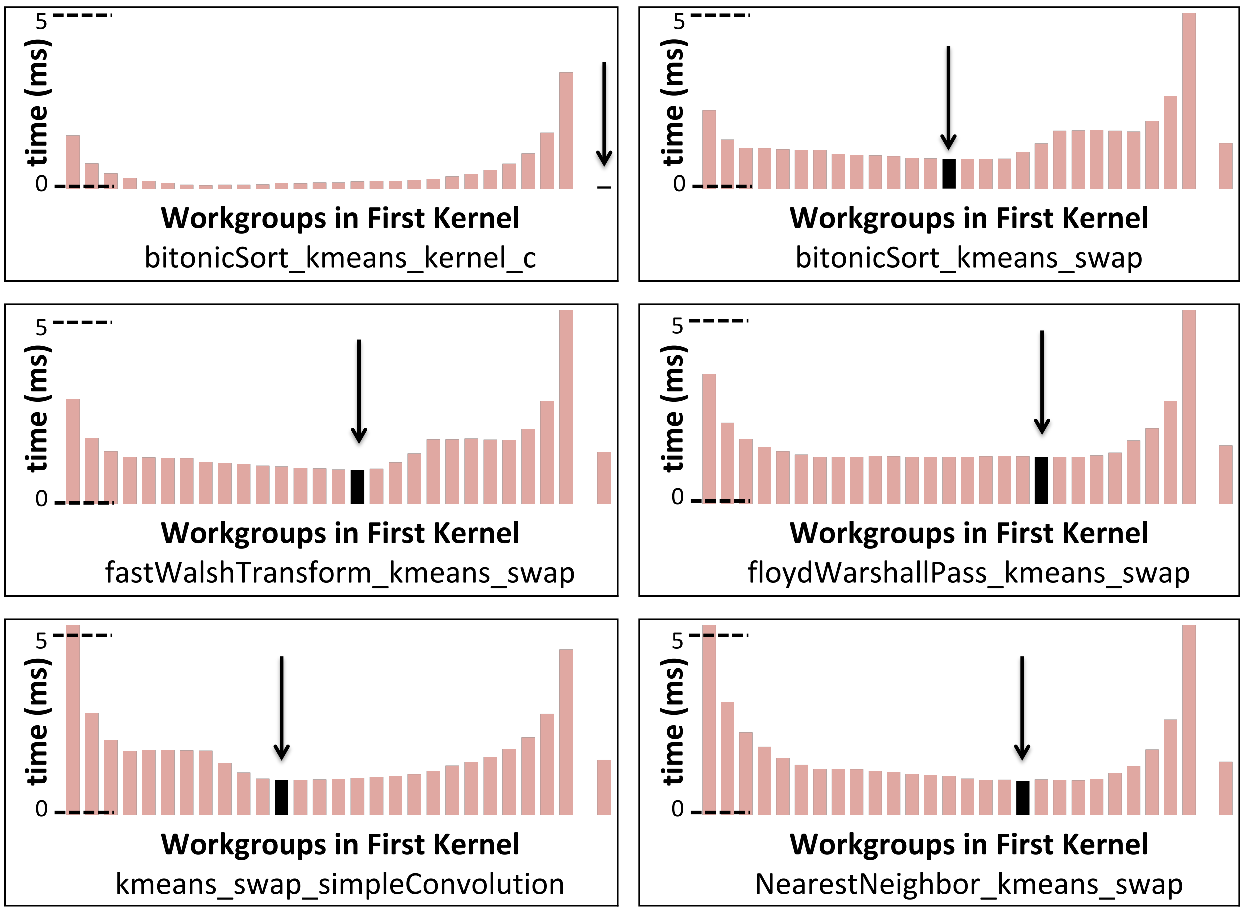


Figure 6.7: Runtimes for concurrent kernels with the partitioning algorithm. The dark column with the arrow indicates the minimum value. The final column indicates the combined sequential runtime for each merged kernel pair.

the optimal results found by the partitioning algorithm, without needing to perform an exhaustive search.

6.4 Observations

We have presented a justification for our argument that running concurrent kernels on a GPU can be beneficial for overall compute throughput. We also showed that naïvely scheduling kernel pairs together can have a negative effect, and therefore it is necessary to dynamically schedule kernels together. We demonstrated *KernelMerge*, an OpenCL concurrent kernel scheduler that has a number of dials important for concurrent kernel research. We have

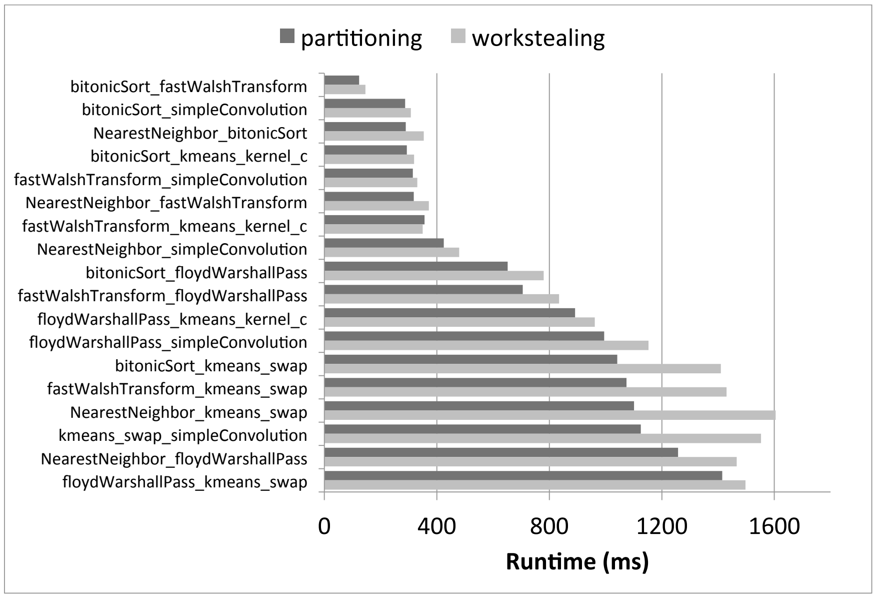


Figure 6.8: Results of the work-stealing scheduler compared to the sweet spot partitioning scheduler result. In most cases, the work-stealing algorithm approaches the partitioning sweet spot.

also demonstrated that a work-stealing algorithm for merging two kernels can approximate optimal workgroup partitioning.

Chapter 7

Related Work

Related work in the field of scheduling for task scheduling on multiple processors can be separated into work on scheduling for homogeneous multiprocessor systems and scheduling for heterogeneous multiprocessor systems. Furthermore, both types of scheduling also have subcategories that include distributed systems for which the processors are allocated to multiple computers on a network and local single-computer machines with multiple processors that share a bus. Sections 7.1 and 7.2 describe the related work in each category as it applies to the type of scheduling described in this dissertation, and Section 7.2 is further broken down to include both distributed systems and General Purpose Graphics Processing Unit (GPGPU) systems. The remaining sections of this chapter address in more detail the related work for Chapters 3 – 6.

7.1 Scheduling for homogeneous multiprocessor systems

Scheduling research for homogeneous multiprocessor systems mostly focuses on individual thread scheduling in an attempt to balance computational load across the available processors. The biggest difference between scheduling for homogeneous systems and heterogeneous systems is that in the former there is an implied assumption that a thread will behave the same regardless of the processor, and important scheduling concerns focus on load balancing and cache usage. The research most closely related to the work for this dissertation involves resource-aware scheduling for architecturally homogeneous systems, in particular processor resource contention. Fedorova et al. [2010], Parekh et al. [2000], Zhuravlev et al. [2010] and von Behren [2003] monitor thread behavior and make scheduling decisions based on resource requirements per thread. Processor utilization is a key concern in many scheduling methodologies, and many scheduling ideas attempt to utilize all of the available processors as much as possible. Ousterhout [1982] provides a number of straightforward scheduling techniques to improve processor utilization, and Gupta et al. [1991] show that traditional busy-waiting policies need to be modified to enable better processor utilization. The work in this dissertation focuses on high utilization for all the devices in a heterogeneous system, in the effort to provide the maximum computational throughput. It is distinct from the homogeneous scheduling work discussed above in three significant ways. First, processes can be migrated on most homogeneous systems, and this is not the case for today's CPU-GPU systems. Our assumption that a kernel cannot be preempted underlies the scheduling algorithms we discuss in chapter 5. Second, resource contention can be balanced among processors in a homogeneous system without knowing process runtimes. In a heterogeneous system, balancing tasks becomes a matter of understanding the difference in runtime between devices, otherwise a scheduler can easily make a decision that hurts performance. Finally, instruction sets on CPUs and GPUs are incompatible, and an application must be compiled for all available platforms in order to schedule between them.

Task profiling is also frequently mentioned in homogeneous scheduling research. Hoffman and Rauber [2007] discuss task waiting time and its importance in making scheduling decisions. They also discuss making performance predictions and demonstrate waiting time prediction by relying solely on profiling as opposed to looking at program code or binaries, as do Marin and Mellor-Crummey [2004]. The historical data scheduling scheme from Chapter 5 relies on historical profiling instead of application details. Our work is distinct from prior research in that we use historical profiling that compares the runtime on different devices, specifically CPUs and GPUs. We also use a history database of transfer times, which is a novel contribution.

7.2 Scheduling for heterogeneous multiprocessor systems

GPGPU computing is a recent development in the heterogeneous computing arena, but distributed heterogeneous systems have been in production since the early 1990s. Before GPU era, "heterogeneous" generally meant that a single application binary would run at different speeds on different compute nodes, but the nodes would use the same instruction set, and this is not the case with applications that run on both a CPU and a GPU. However, many research papers that discuss distributed heterogeneous scheduling contain applicable ideas to CPU/GPU heterogeneous systems.

Scheduling for Distributed Heterogeneous Systems

As of version 3.0, NVIDIA's CUDA has a limited ability to run concurrent kernels devices with compute capability 2.0 or above (e.g., Fermi GPUs) [NVIDIA, 2010]. The kernels must be run in different CUDA streams that are explicitly set up by the programmer, and the CUDA runtime does not guarantee that kernels will run concurrently. Furthermore, the number of workgroups per kernel that are run concurrently is not definable by the programmer. *KernelMerge* allows workgroup counts to be set in the scheduler, and although there are crucial overheads at this point in time, minimal hardware changes would significantly increase the efficiency of *KernelMerge*.

Chen et al. [2010] propose a dynamic load balancing scheme for single and multiple GPUs. They implement a task queue that has similar functionality to *KernelMerge* and that launches thread blocks (the CUDA equivalent of OpenCL workgroups) independently of the normal CUDA launch behavior. They show a slight increase in performance on a single GPU using their scheduler, due to their scheduler's ability to begin execution of new thread blocks while other thread blocks are still running longer kernels. They do not discuss load balancing as it relates to the type of bounding exhibited by individual kernels, and we discuss this in chapter 6. Wang et al.[2010] propose a source-code level kernel merging scheme in the interest of better device utilization for decreased energy usage. Their tests are limited to GPGPU-Sim [Bakhoda et al., 2009], but their experiments indicate significant energy reduction for fused kernels over serial execution. Our work differs from theirs in that we focused on combining kernels based on how they are bound, not on energy efficiency. Furthermore, we built *KernelMerge* for real hardware, and therefore were able to test the results under actual conditions.

Li et al. [2011] propose a GPU virtualization layer that allows multiple microprocessors to share GPU resources. Their virtualization layer implements concurrent kernels in CUDA, and they discuss computation and memory boundedness (which they call "compute intensive" and "I/O-intensive") as it relates to running kernels concurrently. It is not clear how they classified the individual kernels in order to determine how they were bound. Menasce et al. [1995] and Kafil and Ahmad [1997] were among the first to investigate scheduling for heterogeneous systems, and they discuss both dynamic and static scheduling policies and the benefits and drawbacks of both. Static schedulers have lower (or no) runtime overhead, and can utilize sophisticated algorithms that can run before applications are run. However, they also need to know enough about the runtime profile and input data to make scheduling decisions, and the uncertainty inherent in this model can lead to an unbalanced load. Dynamic scheduling, on the other hand, makes decisions with more knowledge of the system and the application inputs, but incurs extra cost in runtime overhead. These scheduling issues translate directly into the desktop heterogeneous world. Siegel and Ali [2000] and Radulescu and Van Gemundalso [2000] look at the tradeoffs between static and dynamic scheduling for heterogeneous tasks for distributed systems. In particular, Siegel and Ali focus on scheduling independent tasks onto heterogeneous nodes, and they provide a number of heuristics that are applicable to desktop heterogeneous systems. Their *min-min* algorithm attempts to determine the device which will complete a task fastest, and the idea we have presented using historical data for scheduling uses a similar algorithm. Topcuoglu et al. [1999] also describe an algorithm that looks at when applications will finish on a processor. Their *heterogeneous earliest-finish-time* (HEFT) algorithm attempts to minimize when individual applications finish, although they do not discuss how they are gathering the data used for the prediction. Our work is distinct from Siegel and Ali's work and from HEFT in that we developed an algorithm that uses device runtimes as well as data transfer times to make our scheduling decisions.

Scheduling for Heterogeneous Systems

There have been a number of recent papers on heterogeneous scheduling. Jiménez et al. [2009], which was completed in parallel with our work, demonstrate two predictive algorithms that use performance history to schedule a queue of applications between a CPU and a GPU: *history-gpu*, that schedules work on the first available device that can run an application, and *estimate-hist*, that estimates the waiting time for each device and schedules an application to the device that will be free the soonest. Their work does not take into consideration data transfer overhead, nor do they discuss how they make the runtime predictions for their scheduler. The scheduler we present in Chapter 5 schedules kernels on the devices that will enable them to finish first, instead of simply looking at which device will be free soonest, and we also consider data transfer overhead and provide a model for runtime prediction across devices.

Augonnet et al. [2009] use performance models to provide scheduling hints for their StarPU scheduler, and programmers who write applications for StarPU can provide a "cost model" for each application that enables the scheduler to predict relative runtimes. Our method does not require programmers to provide any additional information to the scheduler, and does not require performance models for specific hardware.

Becci et al. [2010] use performance and data transfer overhead history to guide a dynamic heterogeneous scheduler for legacy kernels, focusing on postponing data transfer between devices until it is actually needed. They choose not to migrate data when the overhead is too high, and their scheduling method attempts to move the computation to the data, if that is possible. Our work takes a different approach: we look at data transfer overhead as part of the overall kernel runtime, and account for it explicitly in the scheduling decision, based on a prediction of which device will complete the kernel sooner. Furthermore, our work does not require programmer input, nor does it postpone data transfers.

Other researchers have proposed using performance models to predict runtime, which could be used for making scheduling decisions (e.g., Meng and Skadron [2009] and Hong and Kim [2010]). However, performance models have high overhead and are generally not portable between hardware generations. Shelepov et al. [2009] describe a static heterogeneous scheduler that uses *architectural signatures* to match an application to heterogeneous cores. It does not have to rely on dynamic profiling, making it more effective for workloads that have significantly different run times from one run to the next. Rather, it makes scheduling decisions based specifically on a history database of runtimes, and work is scheduled in the order it is received in the task queue.

The method for heterogeneous scheduling in Chapter 5 looks at entire kernels and schedules them on one of the available devices in a heterogeneous computer. Others have looked at finer-grain scheduling methods, which partition kernels such that one kernel launches across multiple devices. An example is Luk et al. [2009], which also uses a historical database for *Qilin* that holds runtime data for applications it has seen before. Their approach dynamically compiles kernels to run across multiple devices at once, which is distinct from our method of running independent tasks on different devices.

7.3 Data Transfer and Overhead

It is well known that PCI Express bandwidth can cause a throughput bottleneck when a significant amount of data is transferred between a CPU and a GPU in a heterogeneous system. A number of researchers have discussed bandwidth troubles that can arise with frequent or poorly managed data movement between devices. Schaa and Kaeli [2009] examine multiple GPU systems and acknowledge that unless a full working set of data can fit into the memory on a GPU, the PCI Express will be a bottleneck. Owens et al. [2008] express similar concerns. Fan et al. [2004], Cohen and Molemaker [2009] and Dotzler et al. [2010] all recommend rewriting algorithms to limit PCI Express transfers as much as possible. The aforementioned studies served as our motivation for this work, and we decided to quantify the memory transfer overheads for a number of benchmarks and to develop an an overhead taxonomy for discussing the overhead in more detail.

Some researchers have made suggestions about how to mitigate data transfer overhead limitations. Al-Kiswany et al. [2008] describe StoreGPU, a distributed storage system that uses pinned, non-pageable memory on the host system to reduce the impact of data transfer. Gelado et al. [2010] introduce an "Asymmetric Distributed Shared Memory" (ADSM) model that provides two types of memory updates ("Lazy" and "Rolling") that determine when to move data on and off the GPU.

An important GPGPU research area involves migrating database operations to the GPU, and moving all or portions of a database to the GPU involves a significant cost. Bakkum and Skadron [2010] describe a SQL database GPGPU solution that demonstrates a 1.4x performance degradation when considering data transfer overhead in their results, and the entire database must also be located on the device. Volk et al. [2010] implement a GPU database that also requires the entire database to be moved to the GPU in order to keep data transfer at a minimum. Additionally, they recognize that in order to benefit from a GPU solution, they must amortize the cost of moving the database to the device by performing multiple database operations. Govindaraju et al. [2006] use a bitonic sort algorithm to perform database sort operations, and they specifically optimize their code to limit memory transfers between devices due to memory transfer overhead. Our work takes a broader view of the subject of data transfer, and categorizes the required types of data transfer into a comprehensive taxonomy. Whereas other related work looked at individual data transfer requirements related to specific algorithms or problems, our work treats the idea comprehensively and is algorithm-agnostic.

7.4 Runtime Prediction

Most of the research into predicting runtime for heterogeneous processors has targeted distributed heterogeneous systems that consist of CPUs from different hardware generations, and not CPU/GPU systems [Nadeem and Fahringer, 2009; Page, 2009; Iverson et al., 1999]. A number of those papers describe a multidimensional k-NN regression algorithm for predicting task run times based on historical data, and this algorithm can be directly applied to CPU/GPU heterogeneous kernels. Iverson et al. [1999] and Page [2009] proposed modeling heterogeneous task inputs as a vector of random variables and applying a multidimensional k-NN regression algorithm to new inputs in order to predict task runtimes for new inputs. Their results demonstrate the precise nature of the algorithm, but they do not discuss the time required to make the prediction calculations. Our work is the first to provide a combined runtime prediction model that dynamically adjusts (and improves) as more runtime data is collected. In chapter 4 we demonstrate the improvement of our algorithms over using k-NN and linear prediction alone.

The use of performance models to predict heterogeneous runtimes have also been discussed in the literature (e.g., Meng and Skadron [2009] and Hong et al. [2010]), but performance models also have high overhead and must be redesigned for each new hardware generation.

A number of researchers have proposed using historical data in order to make runtime predictions for heterogeneous scheduling decisions [Jiménez et al., 2009; Augonnet et al., 2009;

Diamos and Yalamanchili, 2008; Becchi et al., 2010; Topcuoglu et al., 1999], but they do not discuss how precise the predictions need to be in order to be useful for their scheduling algorithms, nor do they discuss how those predictions are calculated. However, Armstrong et al. [1998] investigated four task mapping scheduling algorithms based on expected runtimes and found that exact runtime prediction was not necessary for sufficient task mapping, even when the predictions showed significant variance. These results motivate our reasoning that it is better to provide a faster runtime prediction that sacrifices precision than to take more time to refine the prediction results. Our work provides an algorithm that could be used in the above cited works in order to make the runtime predictions.

7.5 Dynamic Scheduling Algorithms

The idea of using historical data for heterogeneous scheduling decisions has been discussed in other work, although very few have targeted GPU computing. Siegel et al. [1996] discuss automated heterogeneous scheduling where one stage profiles tasks and performs performs analytical benchmarking of individual tasks. This information is then used in a later stage to predict runtimes for kernels based on current processor loads. Similarly, our approach profiles kernels as they run, but also extrapolates runtimes for kernels with different input sizes from an analytical assessment of the collected data. Topcuoglu et al. describe the "Heterogeneous Earliest-Finish-Time" (HEFT) algorithm [Topcuoglu et al., 1999], which, like our approach, attempts to minimize when individual kernels finish, but does not consider fairness or make any data transfer considerations. Maheswaran et al. [1999] describe a set of heuristics that inform a dynamic heterogeneous scheduler. Their *Min-min* heuristic calculates which device will provide the earliest expected completion time across a set of tasks on a system. The task in the queue that will complete first is scheduled next. Both works of Topcuoglu et al. and Maheswaran et al. were written prior to the advent of GPU computing, and they simulated their algorithms. Our approach differs from both Topcuoglu et al. and Maheswaran et al. by considering fairness, ensuring that kernels do not get pre-empted by other kernels, and we also tested our scheduler on CPU/GPU heterogeneous hardware.

Harmony [Diamos and Yalamanchili, 2008] also uses performance estimates in order to schedule kernels across a heterogeneous system. They propose an online monitoring of kernels and describe a dependence-driven scheduling that analyzes how kernels share data and decides on processor allocation based on which applications can run without blocking. Our approach considers kernels to be independent, and schedules kernels from multiple applications concurrently.

7.6 Kernel Merging

As of version 3.0, NVIDIA's CUDA has a limited ability to run concurrent kernels devices with compute capability 2.0 or above (e.g., Fermi GPUs) [NVIDIA, 2010]. The kernels must be run in different CUDA streams that are explicitly set up by the programmer, and the CUDA runtime does not guarantee that kernels will run concurrently. Furthermore, the number of workgroups per kernel that are run concurrently is not definable by the progarmmer. Our main contribution, *KernelMerge* (Chapter 6), allows workgroup counts to be set in the scheduler, and although there are crucial overheads at this point in time, minimal hardware changes would significantly increase the efficiency of *KernelMerge* [Gregg et al., 2012].

Guevara et al. [2009] discuss a CUDA scheduler that runs orthogonally bound kernels to increase throughput for both kernels, although all kernels must be hand-merged and analyzed by hand. Wang et al. [2011] demonstrate the benefits of kernel "funneling" of CUDA kernels into a context so that they can be run concurrently. They do not discuss how the kernels interfere with each other on a GPU, although their benchmarks are most likely memory bound as they perform relatively few calculations per memory access.

Chapter 8

Conclusions

This dissertation has characterized a dynamic heterogeneous scheduling methodology that increases computational throughput when a set of applications are run on a computer with a multicore CPU and one or more GPUs and when each application has the ability to run on either type of device. The dissertation presents a taxonomy that encapsulates data transfer requirements for kernels based on the location of data prior to running a kernel, whether the data will need to be moved prior to or after the kernel runs, and whether the data is created or consumed during kernel execution. The dissertation also describes a novel hybrid runtime prediction model to provide precise and rapid kernel runtime forecasts for use in a heterogeneous scheduler. The prediction model, along with data locality and contention information, is used in a prototype dynamic scheduler that can produce schedules that are 33% faster than GPU-only schedules and provide 90% CPU utilization. Finally, the dissertation demonstrates that orthogonally bound kernels (i.e., memory bound and compute bound) can be run concurrently on a GPU and can show improved throughput compared to running the kernels sequentially.

Heterogeneous scheduling for CPU/GPU systems will continue to become more important as more application developers write general purpose code for GPUs, and as they use programming languages that are agnostic towards the target device for their programs.

Current scheduling techniques are insufficient when multiple applications wish to use the GPU for general purpose computation, leading to contention for the GPU and underutilization of the CPU. Scheduling for a heterogeneous computer is not trivial, and we have shown that it requires knowledge of data locality and must be able to make runtime predictions for applications on each device.

8.1 Impact

Operating system schedulers have become efficient at scheduling work for multiple homogeneous cores, especially at the thread-level. However, with the advent of general purpose computing for GPUs, desktop computers are no longer homogeneous, and operating schedulers do not schedule work for heterogeneous cores. If an application developer wants general purpose work completed on a GPU, he or she must explicitly schedule it for that device, and it may not be the best device to complete the work, especially given data locality and contention concerns. Heterogeneous scheduling has all of the demands of homogeneous scheduling plus its own set of challenges. This research has demonstrated dynamic scheduling in heterogeneous systems using a combination of historical data, statistical information, and environment state information, and frees application developers from manually performing the scheduling. Furthermore, this dynamic scheduling results in greater processing throughput than in a statically scheduled system. The impact of the findings in this dissertation provide a foundation for dynamic heterogeneous scheduler improvements, and will be necessary for efficient heterogeneous scheduling. The following subsections describe this dissertation's impact on specific aspects of the heterogeneous scheduling problem.

8.1.1 Data Location and Transfer Overhead

Discrete GPUs have an independent memory system from the CPU, and in order to perform computation on a set of data, the data must be located in GPU memory. In many cases data must be transferred between devices as an application progresses. A scheduler that ignores this overhead will not be able to schedule the work appropriately in cases where the data transfer time is large enough to negate the benefits of running on a device that can complete the work faster. If significant data needs to be transferred either to or from a device, the transfer times must be considered in order to make a valuable scheduling decision. This research describes a novel classification scheme for describing memory transfer requirements for individual kernels, and makes suggestions for properly utilizing this information in a dynamic scheduler and when reporting on and comparing runtimes between different devices.

The advent of integrated CPU-GPU processors such as AMD's Accelerated Processing Unit (APU) systems will modify the impact of the data locality components of this dissertation. In particular, single-memory systems will not suffer from slowdowns related to moving the data between processor memories. However, both the CPU and GPU will be competing for the same memory system, and therefore similar arguments about where data is located will have to be considered. Contention for memory controllers, and for bandwidth on the memory bus will share some of the same memory transfer issues that we considered for this dissertation. For example, it may be the case that data for a particular application should be spread out across a memory system so that one memory controller does not have bottlenecks when attempting to deliver a large amount of data to the GPU processing elements. In principle, this is the same problem we tackled with moving data between the CPU to the GPU, except that it is within one memory system.

A secondary impact of the classification scheme is the ability to use the taxonomy when comparing application runtimes between CPUs and GPUs. Without considering data transfer, simple runtime comparisons provide results that are not useful for real world systems, and are disingenuous. It may be the case that a GPU can run a kernel ten or twenty times faster than a CPU, but if the data needs to be transferred to and from the GPU, the runtime results by themselves do not provide a fair comparison. In order to make a fair comparison about the usefulness of using a GPU for a particular application, data transfer use cases must be included.

8.1.2 The Necessity of Precise and Rapid Runtime Prediction

In a CPU/GPU heterogeneous computer, applications that can run on both devices typically have significantly different runtimes. In order to build a schedule for such a system, the scheduler must be able to predict runtimes for each application on each device. We have demonstrated that this can be accomplished with the use of historical runtime information, and we have described a novel fast runtime prediction model. The ability to predict runtimes is necessary for a heterogeneous scheduler for two primary reasons. First, because runtimes can be drastically different, schedules that ignore runtimes run the risk of low device utilization and poor overall throughput across the tasks. Second, load balancing on a set of homogeneous CPUs can be achieved by migrating processes between devices and by time slicing processes on a single device. Neither is possible on GPUs, with the ramification that once a process begins on a GPU, it must complete before another application can be scheduled on that device. Precise runtime prediction provides the scheduler with important information to improve scheduling decisions.

A second impact of the hybrid prediction model is to demonstrate the tradeoffs for precise and fast prediction. The time to make scheduling decisions is an important consideration, especially when kernel runtimes are on the order of microseconds or milliseconds. Benefits from a precise prediction can be lost if the time to make the prediction contributes significantly to a decrease in throughput. The hybrid prediction model described in this dissertation has demonstrated the ability to control the knob for prediction precision and speed.

The impact that runtime prediction provides to heterogeneous scheduling will change if GPU kernel pre-emption and kernel migration are implemented into future devices, however it will still provide important information to a scheduler. For example, migration will always incur overhead costs, and a scheduling decision that limits the amount of migration will lessen the overhead, and have a positive impact on overall throughput.

8.1.3 Workgroup Level Granularity for GPU Scheduling

KernelMerge (Chapter 6) demonstrates the benefit of scheduling two GPU kernels with a workgroup level granularity. Two kernels that tax separate architectural components of the GPU can run concurrently and have the potential to complete faster than if each kernel were run sequentially. The impact of workgroup level GPU scheduling goes beyond running orthogonally bound kernels, however. Being able to schedule multiple kernels concurrently may not always improve throughput for each kernel, however it could benefit a dynamic heterogeneous scheduler that needs to schedule multiple applications. The scheduler described in Chapter 5, for instance, would have greater scheduling flexibility if more than one kernel could be scheduled onto a GPU. Furthermore, long-running and short-running kernels could potentially be run concurrently, thereby enhancing the ability for a scheduler to ensure that short running jobs are handled efficiently. Finally, as operating system schedulers already schedule multiple applications on one device at a time (generally through time-slicing), integrating GPU scheduling into an OS scheduler will be easier with the ability to concurrently run multiple GPU applications.

8.1.4 The Role of the Heterogeneous Scheduler

Heterogeneous scheduling can take on a variety of forms, both static and dynamic, and from within individual applications, from a runtime separate from the operating system itself, or from within the operating system. Arguably, the operating system has the best information about system state and can provide the most precise scheduling, but current OS schedulers are optimized for scheduling work across homogeneous processors and only consider devices that are identical. This work demonstrates that a runtime scheduler for heterogeneous devices can complement the OS scheduler to make decisions that do take into consideration information specific to a heterogeneous system.

This work can be directly integrated into an operating system scheduler. We expect that this type of OS scheduler evolution is the next step in more throughly uniting CPUs and GPUs in a single system. This could happen most efficiently if GPU vendors collaborate with OS vendors to consider hardware changes that would improve the integration, in much the same way CPU and OS vendors worked together to make changes for better operating system virtualization [Adams and Agesen, 2006]. For example, the ability to stop a GPU process and to report on its current state or progress would drastically improve a scheduler's ability to predict when a device will become free. Likewise, being able to easily run multiple kernels concurrently on a device, or to be able to migrate processes between heterogeneous devices would improve load balancing.

8.2 Future Work

Dynamic heterogeneous scheduling will become even more critical in the next five to ten years, as more developers take advantage of the general purpose capabilities of GPUs. More applications will request the use of GPU resources, and therefore a method for dynamically scheduling work for GPUs is imperative. Multicore scheduling became essential as single-core computers disappeared from the marketplace; powerful GPUs are prevalent on computers already, and processor heterogeneity will continue to proliferate on commodity computers in the next decade. This work prompts continued exploration of heterogeneous scheduling methods, and further investigation on increasing the general purpose utilization of all of the processors, CPU and GPU, available in a system.

We will discuss two avenues for future work related to dynamic heterogeneous scheduling: integration into operating system schedulers, and dynamic kernel merging for concurrent GPU kernels. We will also examine specific GPU features as announced by AMD and NVIDIA, and will discuss the ramifications of future trends to the work in this dissertation.

8.2.1 Heterogeneous Scheduling in the OS Scheduler

As described in Section 8.1.4, many of the results of this dissertation could be integrated into an operating system scheduler that is enhanced to include dynamic scheduling for GPUs as well as CPUs. Mainstream OS schedulers (e.g., on Linux, Mac OS X, and Windows) only schedule for homogeneous CPU systems. All scheduling for GPUs is done through GPU vendor supplied drivers and is coordinated by individual GPGPU application developers. Although it would entail considerable changes to current schedulers, and would almost certainly necessitate substantial collaboration with GPU vendors, the possibility of a mainstream operating system that could dynamically and automatically schedule work across CPUs and GPUs is the next step in furthering GPGPU computing.

An OS scheduler that has the ability to schedule for all computing devices on a system would significantly increase the worth of the GPU to an equipped system. Application developers would be relieved of targeting for a specific device, although it is feasible to leave the ability to preferentially target a GPU or CPU, with the understanding that the scheduler would schedule dynamically based on the best device at the time. The OS scheduler would deconflict contention among applications that would otherwise compete for a particular device, allowing device utilization to improve across all devices.

As of May, 2012, both AMD and NVIDIA have indicated that they would welcome better integration of GPU scheduling into next generation operating systems [AMD, 2012b; NVIDIA, 2012a]. Both vendors foresee a tighter fusion of task-based scheduling for heterogeneous devices, and AMD has introduced the "Fusion System Architecture," (FSA) which is a task-based queuing model that they will implement in the future. In particular, they have described FSA as having a task dispatch model that closely resembles the ideas we have presented for our scheduler. Applications will send generic work to a scheduler, which will determine which of the available devices to run each task on, and will queue the work for each chosen device. The decision about where to run the work can be determined using the ideas we have presented in this dissertation, and efficient task queueing will continue to be an important research area.

8.2.2 Dynamic Kernel Merging for Concurrent GPU Kernels

Chapter 6 describes running concurrent GPU kernels at workgroup level granularity. The next step in this work is to describe a prediction model that enables dynamic merging to increase throughput for multiple kernels. One method that we have already done some preliminary work on mimics the *Bubble-Up* characterization model described by Mars et al. for use in warehouse scale computing systems [2011]. *Bubble-Up* measures the sensitivity of a CPU application to memory system pressure, and also characterizes the contentiousness of the application by how much pressure it places on the memory system. Likewise, a GPU version of *Bubble-Up* would run concurrently with other GPU kernels to provide characterization and sensitivity information. *GPU Bubble-Up* would stress the GPU memory system by continuously reading from global memory, and using the *KernelMerge*, *GPU Bubble-Up* could be tailored to produce a varying amount of memory pressure. “Bubble scores,” which indicate how sensitive applications are to memory pressure, would be subsequently created for individual applications. The bubble scores would indicate whether an application is memory bound or compute bound, and could be integrated into a dynamic scheduler that would concurrently run kernels that have orthogonal boundedness. The results from Chapter 6 demonstrate the promise that such a scheduler could improve throughput for a suitably diverse set of GPU kernels.

8.2.3 GPU Trends and Heterogeneous Scheduling

Both AMD and NVIDIA have made recent announcements that include information about their future plans for scheduling on their devices. This includes a greater ability for concurrent execution of independent tasks on a device, and the possibility of future devices that include context switching and interrupt handling [AMD, 2012b; NVIDIA, 2012a]. These trends will vastly improve the outlook for the conclusions we have drawn in this work; namely, having the ability to schedule concurrent work on a GPU increases the possibility of better overall device utilization, and higher task throughput. Furthermore, a scheduler that can request task switching on a GPU has the potential to time-slice work sent to the device, although it remains to be seen whether this is an optimal solution given the memory transfer overheads we have described.

NVIDIA has released the first version of the *Kepler* line of discrete GPUs, which promise the ability to have multiple CPU cores launch concurrent work on a GPU [NVIDIA, 2012b]. In a task-based queue scheduler like the one we present in chapter 5, being able to schedule concurrent work for a device presents scheduling challenges beyond those that we discussed in this dissertation. In order to maintain the fairness of the schedule as we have described it, kernels must be carefully launched in order to guarantee that they will finish before they would have if run on other devices in the system. Kernels that are concurrently scheduled may be slowed down by virtue of running together, and therefore it will be more difficult to use runtime history to make scheduling predictions. However, overall task throughput could be improved by concurrent scheduling, and more research would need to be completed to make that determination. Our work on kernel boundedness for chapter 6 is particularly applicable to the question of whether or not two or more kernels should be run concurrently on a given device.

AMD has recently described their outlook on the future of what they call “Command and Dispatch Flow” for heterogeneous scheduling in their Fusion System Architecture (FSA) [AMD, 2012b]. In their model, task scheduling is similar to what we have proposed and will employ user-based task queueing that will be scheduled for available devices. Our work on dynamic scheduling will remain relevant because their proposed model must make a determination on which processors will run which tasks, as we discussed in chapter 5.

Bibliography

[Adams and Agesen, 2006] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–13, San Jose, California, 2006.

[Al-Kiswany et al., 2008] Samer Al-Kiswany, Abdullah Gharaibeh, Elizeu Santos-Neto, George Yuan, and Matei Ripeanu. StoreGPU: Exploiting graphics processing units to accelerate distributed storage systems. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, pages 165–174, Boston, MA, June 2008.

[AMD, 2010] AMD. Coming soon: The AMD Fusion family of APUs. http://sites.amd.com/us/fusion/apu/Pages/fusion.aspx. Accessed on October 31, 2010.

[AMD, 2012a] AMD. ATI stream computing compute abstraction layer (CAL). http://developer.amd.com/gpu_assets/ATI_Stream_SDK_CAL_Programming_Guide_v2.0.pdf. Accessed on March 1, 2012.

[AMD, 2012b] AMD. The programmer's guide to the APU galaxy, phil rogers, corporate fellow, AMD. http://developer.amd.com/afds/assets/keynotes/Phil%20Rogers%20Keynote-FINAL.pdf. Accessed on May 25, 2012.

[Armstrong et al., 1998] Robert Armstrong, Debra Hensgen, and Taylor Kidd. The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions. *Seventh Heterogeneous Computing Workshop*, pages 79–87, March 1998.

[Augonnet et al., 2009] C. Augonnet, S. Thibault, R. Namyst, and P.A. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. In *Euro-Par Parallel Processing*, pages 863–874, Delft, The Netherlands, August 2009. Springer.

[Bakhoda et al., 2009] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174, April 2009.

[Bakkum and Skadron, 2010] Peter Bakkum and Kevin Skadron. Accelerating SQL database operations on a GPU with CUDA. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 94–103, 2010.

[Becchi et al., 2010] Michela Becchi, Surendra Byna, Srihari Cadambi, and Srimat Chakradhar. Data-aware scheduling of legacy kernels on heterogeneous platforms with distributed memory. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 82–91, June 2010.

[Bell and Garland, 2009] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, Portland, OR, November 2009.

[Che et al., 2008] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing*, 68(10):1370–1380, October 2008.

[Che et al., 2009] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. *IEEE Workload Characterization Symposium*, pages 44–54, October 2009.

[Chen et al., 2010] Long Chen, O. Villa, S. Krishnamoorthy, and G.R. Gao. Dynamic load balancing on single- and multi-GPU systems. In *2010 IEEE International Symposium on Parallel Distributed Processing*, pages 1–12, april 2010.

[Cohen and Molemaker, 2009] J. Cohen and M.J. Molemaker. A fast double precision CFD code using CUDA. In *Parallel Computational Fluid Dynamics: Recent Advances and Future Directions*, pages 414–429, Moffett Field, CA, May 2009.

[Diamos and Yalamanchili, 2008] Gregory F. Diamos and Sudhakar Yalamanchili. Harmony: an execution model and runtime for heterogeneous many core systems. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, pages 197–200, 2008.

[Dotzler et al., 2010] Georg Dotzler, Ronald Veldema, and Michael Klemm. JCUDAmp: OpenMP/Java on CUDA. In *3rd International Workshop on Multicore Software Engineering*, pages 10–17, May 2010.

[England, 1978] J. N. England. A system for interactive modeling of physical curved surface objects. In *Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH, pages 336–340, 1978.

[Fan et al., 2004] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. GPU cluster for high performance computing. In *ACM/IEEE Conference on Supercomputing*, pages 47–58, Pittsburgh, PA, November 2004.

[Fedorova et al., 2010] Alexandra Fedorova, Sergey Blagodurov, and Sergey Zhuravlev. Managing contention for shared resources on multicore processors. *Communications of the ACM*, 53(2):49–57, 2010.

[Flynn, 1966] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, December 1966.

[Fürst et al., 2011] Norbert Fürst, Armin Weiß, Martin Heide, Simon Papandreou, and Ana Balevic. CUJ2K. `http://cuj2k.sourceforge.net/cuj2k.html`. Accessed on February 21, 2011.

[Gelado et al., 2010] Isaac Gelado, John E. Stone, Javier Cabezas, Sanjay Patel, Nacho Navarro, and Wen-mei W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. In *Architectural Support for Programming Languages and Operating Systems*, pages 347–358, Pittsburgh, PA, March 2010.

[Google, 2012] Google. Google scholar. http://scholar.google.com. Accessed on March 5, 2012.

[Govindaraju et al., 2006] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPUTeraSort: High performance graphics co-processor sorting for large database management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 325–336, June 2006.

[Gregg and Hazelwood, 2011] Chris Gregg and Kim Hazelwood. Where is the data? Why you cannot debate GPU vs. CPU performance without the answer. In *International Symposium on Performance Analysis of Systems and Software (ISPASS).*, April 2011.

[Gregg et al., 2010] C. Gregg, J. Brantley, and K. Hazelwood. Contention-aware scheduling of parallel code for heterogeneous systems. In *2nd USENIX Workshop on Hot Topics in Parallelism*, Berkeley, CA, June 2010.

[Gregg et al., 2011] C. Gregg, M. Boyer, K. Hazelwood, and K. Skadron. Dynamic heterogeneous scheduling decisions using historical runtime data. In *2nd Workshop on Applications for Multi- and Many-Core Processors*, San Jose, CA, June 2011.

[Gregg et al., 2012] C. Gregg, J. Dorn, K. Skadron, and K. Hazelwood. Fine-grained resource sharing for concurrent GPGPU kernels. In *24th USENIX Workshop on Hot Topics in Parallelism*, Berkeley, CA, June 2012.

[Guevara et al., 2009] M. Guevara, C. Gregg, K. Hazelwood, and K. Skadron. Enabling task parallelism in the CUDA scheduler. In *Programming Models and Emerging Architectures Workshop (Parallel Architectures and Compilation Techniques Conference)*, 2009.

[Gupta et al., 1991] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. The impact of operating system scheduling policies and synchronization methods of performance of parallel applications. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and modeling of computer systems*, pages 120–132, San Diego, CA, May 1991.

[Hoberock et al., 2009] Jared Hoberock, Victor Lu, Yuntao Jia, and John C. Hart. Stream compaction for deferred shading. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 173–180, August 2009.

[Hoffmann and Rauber, 2007] R. Hoffmann and T. Rauber. Profiling of task-based applications on shared memory machines: Scalability and bottlenecks. *Lecture Notes in Computer Science*, 4641:118, 2007.

[Hong and Kim, 2010] Sunpyo Hong and Hyesoon Kim. An integrated GPU power and performance model. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, pages 280–289, 2010.

[Iverson et al., 1999] M.A. Iverson, F. Ozguner, and L.C. Potter. Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment. In *Eighth Heterogeneous Computing Workshop.*, pages 99–111, San Juan, Puerto Rico, April 1999.

[Jiménez et al., 2009] Víctor J. Jiménez, Lluís Vilanova, Isaac Gelado, Marisa Gil, Grigori Fursin, and Nacho Navarro. Predictive runtime code scheduling for heterogeneous architectures. In *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, pages 19–33, Berlin, Heidelberg, 2009. Springer-Verlag.

[Kafil and Ahmad, 1997] M. Kafil and I. Ahmad. Optimal task assignment in heterogeneous computing systems. In *6th Heterogeneous Computing Workshop*, pages 135–146, 1997.

[Kindratenko et al., 2009] V.V. Kindratenko, J.J. Enos, Guochun Shi, M.T. Showerman, G.W. Arnold, J.E. Stone, J.C. Phillips, and Wen mei Hwu. GPU clusters for high-performance computing. In *IEEE International Conference on Cluster Computing and Workshops*, pages 1–8, New Orleans, LA, sept 2009.

[Lee et al., 2010a] D. Lee, I. Dinov, B. Dong, B. Gutman, I. Yanovsky, and A.W. Toga. CUDA optimization strategies for compute-and memory-bound neuroimaging algorithms. *Computer Methods and Programs in Biomedicine*, pages 1–13, December 2010.

[Lee et al., 2010b] V.W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A.D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, and P. Hammarlund. Debunking the 100x GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU. In *37th Annual International Symposium on Computer Architecture*, pages 451–460, Saint-Malo, France, June 2010.

[Li et al., 2011] Teng Li, V.K. Narayana, E. El-Araby, and T. El-Ghazawi. GPU resource sharing and virtualization on high performance computing systems. In *2011 International Conference on Parallel Processing (ICPP)*, pages 733–742, sept. 2011.

[Luk et al., 2009] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture, 2009.*, pages 45–55, December 2009.

[Maheswaran et al., 1999] Muthucumaru Maheswaran, Shoukat Ali, Howard Jay Siegel, Debra Hensgen, and Richard F. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. *Heterogeneous Computing Workshop*, 0:30, 1999.

[Marin and Mellor-Crummey, 2004] Gabriel Marin and John Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. *SIGMETRICS Performance Evaluation Review*, 32:2–13, June 2004.

[Mars et al., 2011] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing sensible co-locations for improved utilization in modern warehouse scale computers. In *Proceedings of The 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011.

[Menasce et al., 1995] D.A. Menasce, D. Saha, S.C.S. Porto, V. Almeida, and S.K. Tripathi. Static and dynamic processor scheduling disciplines in heterogeneous parallel architectures. *Journal of Parallel and Distributed Computing*, 28(1):1–18, 1995.

[Meng and Skadron, 2009] Jiayuan Meng and Kevin Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In *Proceedings of the 23rd International Conference on Supercomputing*, pages 256–265, November 2009.

[Merrill and Grimshaw, 2010] Duane G. Merrill and Andrew S. Grimshaw. Revisiting sorting for gpgpu stream architectures. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 545–546, 2010.

[Morgan, 2012] Timothy Morgan. Amazon adds GPUs to EC2 HPC clouds. http://www.theregister.co.uk/2010/11/15/amazon_ec2_gpu_slices/. The Register. Accessed on February 20, 2012.

[Nadeem and Fahringer, 2009] Farrukh Nadeem and Thomas Fahringer. Predicting the execution time of grid workflow applications through local learning. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, Portland, OR, November 2009.

[NVIDIA, 2010] CUDA Programming Guide 3.1. http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf, June 2010.

[NVIDIA, 2011a] NVIDIA. CUBLAS library 3.1. http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/CUBLAS_Library_3.1.pdf. Accessed on February 21, 2011.

[NVIDIA, 2011b] NVIDIA. CUDA programming guide, version 3.1. http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_{CUDA}_C_ProgrammingGuide_3.1.pdf. Accessed on February 21, July 2011.

[NVIDIA, 2011c] NVIDIA. CUFFT library 3.0. http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/CUFFT_Library_3.0.pdf. Accessed on February 21, 2011.

[NVIDIA, 2012a] NVIDIA. Jen hsun-huang keynote, GTC 2012. http://blogs.nvidia.com/2012/05/behind-the-scenes-at-gtc-2012-opening-keynote/jen-hsun-huang-keynote-gtc-2012/. Accessed on May 25, 2012.

[NVIDIA, 2012b] NVIDIA. NVIDIA kepler GK110 next-generation CUDA compute architecture. http://www.nvidia.com/content/PDF/kepler/NV_DS_Tesla_KCompute_Arch_May_2012_LR.pdf. Accessed on July 1, 2012.

[NVIDIA, 2012c] NVIDIA. OpenCL optimization guide. http://developer.download.nvidia.com/CUDA/training/NVIDIA_GPU_Computing_Webinars_Best_Practises_For_OpenCL_Programming.pdf. Accessed on May 16, May 2012.

[Ousterhout, 1982] J.K. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pages 22–30, Miami, FL, October 1982.

[Owens et al., 2008] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[Page, 2009] A.J. Page. *Adaptive Scheduling in Heterogeneous Distributed Computing Systems*. Phd thesis, National University of Ireland, Galway, Ireland, 2009.

[Parekh et al., 2000] S. Parekh, S. Eggers, H. Levy, and J. Lo. Thread-sensitive scheduling for SMT processors. *University of Washington Technical Report*, pages 04–02, 2000.

[Podlozhnyuk and Harris, 2008] V. Podlozhnyuk and M. Harris. Monte carlo option pricing. *NVIDIA Corporation Tutorial*, 2008.

[Podlozhnyuk, 2007] V. Podlozhnyuk. Image convolution with CUDA. *NVIDIA Corporation white paper, June*, 2097(3), 2007.

[Podlozhnyuk, 2011] V. Podlozhnyuk. Histogram calculation in CUDA. http://www.nvidia.com/object/cuda_sample_data-parallel.html. Accessed on June 7, 2011.

[Radulescu and Gemund, 2000] Andrei Radulescu and Arjan J.C. Van Gemund. Fast and effective task scheduling in heterogeneous systems. *Heterogeneous Computing Workshop*, 0:229, 2000.

[Ryoo et al., 2008] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 73–82, 2008.

[Schaa and Kaeli, 2009] D. Schaa and D. Kaeli. Exploring the multiple-GPU design space. In *International Parallel and Distributed Processing Symposium*, pages 1–12, May 2009.

[Shelepov et al., 2009] Daniel Shelepov, Juan Carlos Saez Alcaide, Stacey Jeffery, Alexandra Fedorova, Nestor Perez, Zhi Feng Huang, Sergey Blagodurov, and Viren Kumar. HASS: a scheduler for heterogeneous multicore systems. *SIGOPS Operating System Review*, 43(2):66–75, 2009.

[Siegel and Ali, 2000] H.J. Siegel and S. Ali. Techniques for mapping tasks to machines in heterogeneous computing systems. *Journal of Systems Architecture*, 46:627–639, 2000.

[Siegel et al., 1996] Howard Jay Siegel, Henry G. Dietz, and John K. Antonio. Software support for heterogeneous computing. *ACM Computing Surveys*, 28(1):237–239, 1996.

[Stone et al., 2010] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science and Engineering*, 12:66–73, 2010.

[Stratton et al., 2008] John Stratton, Sam Stone, and Wen-mei Hwu. MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. In *Languages and Compilers for Parallel Computing*, volume 5335 of *Lecture Notes in Computer Science*, pages 16–30. Springer Berlin / Heidelberg, 2008.

[Supercomputing Blog, 2010] Search algorithm with CUDA. http://supercomputingblog.com/cuda/search-algorithm-with-cuda/. Accessed on July 28, 2010.

[Topcuoglu et al., 1999] H. Topcuoglu, S. Hariri, and Min-You Wu. Task scheduling algorithms for heterogeneous processors. In *8th Heterogeneous Computing Workshop*, pages 3–14, 1999.

[Volk et al., 2010] P.B. Volk, D. Habich, and W. Lehner. GPU-based speculative query processing for database operations. In *Proceedings of the 1st International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, Singapore, September 2010.

[von Behren et al., 2003] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: scalable threads for internet services. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 268–281, Bolton Landing, NY, October 2003.

[Wang et al., 2010] Guibin Wang, YiSong Lin, and Wei Yi. Kernel fusion: An effective method for better power efficiency on multithreaded GPU. In *Green Computing and Communications, 2010 IEEE/ACM Int'l Conference on Cyber, Physical and Social Computing*, pages 344–350, December 2010.

[Wang et al., 2011] Lingyuan Wang, Miaoqing Huang, and T. El-Ghazawi. Exploiting concurrent kernel execution on graphic processing units. In *2011 International Conference on High Performance Computing and Simulation*, pages 24–32, July 2011.

[Williams et al., 2009] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix–vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178–194, 2009.

[Wu and Liu, 2008] Enhua Wu and Youquan Liu. Emerging technology about GPGPU. In *IEEE Asia Pacific Conference on Circuits and Systems*, pages 618–622, Macao, China, December 2008.

[Wu et al., 2009] Ren Wu, Bin Zhang, and Meichun Hsu. Clustering billions of data points using GPUs. In *Proceedings of the Combined Workshops on Unconventional High Performance Computing Workshop Plus Memory Access Workshop*, pages 1–6, 2009.

[Zhuravlev et al., 2010] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ASPLOS 15*, pages 129–142, Pittsburgh, PA, March 2010.