Improving Resource Utilization in Heterogeneous CPU-GPU Systems

---

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

Michael Boyer

May 2013

© 2013 Michael Boyer

Abstract

Graphics processing units (GPUs) have attracted enormous interest over the past decade due to substantial increases in both performance and programmability. Programmers can potentially leverage GPUs for substantial performance gains, but at the cost of significant software engineering effort. In practice, most GPU applications do not effectively utilize all of the available resources in a system: they either fail to use use a resource at all or use a resource to less than its full potential. This underutilization can hurt both performance and energy efficiency. In this dissertation, we address the underutilization of resources in heterogeneous CPU-GPU systems in three different contexts.

First, we address the underutilization of a single GPU by reducing CPU-GPU interaction to improve performance. We use as a case study a computationally-intensive video-tracking application from systems biology. Because of the high cost of CPU-GPU coordination, our initial, straightforward attempts to accelerate this application failed to effectively utilize the GPU. By leveraging some non-obvious optimization strategies, we significantly decreased the amount of CPU-GPU interaction and improved the performance of the GPU implementation by 26x relative to the best CPU implementation. Based on the lessons we learned, we present general guidelines for optimizing GPU applications as well as recommendations for system-level changes that would simplify the development of high-performance GPU applications.

Next, we address underutilization at the system level by using load balancing to improve performance. We propose a dynamic scheduling algorithm that automatically and efficiently

divides the execution of a data-parallel kernel across multiple, possibly heterogeneous GPUs. We show that our scheduler can nearly match the performance of an unrealistic static scheduler when device performance is fixed, and can provide better performance when device performance varies.

Finally, we address underutilization within a GPU by using frequency scaling to improve energy efficiency. We propose a novel algorithm for predicting the energy-optimal GPU clock frequencies for an arbitrary kernel. Using power measurements from real systems, we demonstrate that our algorithm improves significantly on the state of the art across multiple generations of GPUs. We also propose and evaluate techniques for decreasing the CPU's energy consumption during GPU computation.

Many of the techniques presented in this dissertation can be used to improve the performance and energy efficiency of GPU applications with no programmer effort or software modifications required. As the diversity of available hardware systems continues to increase, automatic techniques such as these will become critical for software to fully realize the benefits of future hardware improvements.

Approval Sheet

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

Doctor of Philosophy (Computer Engineering)

Michael Boyer

Michael Boyer

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

Kevin Skadron

Kevin Skadron, Advisor

Sudhanva Gurumurthi

Sudhanva Gurumurthi, Committee Chair

Jack Davidson

Jack Davidson

Mircean Stan

Mircean Stan

Westley Weimer

Westley Weimer

Accepted for the School of Engineering and Applied Science:

James H. Aylor

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

May 2013

Acknowledgments

The work presented in this dissertation, as well as other work completed during my graduate career, would have not have been possible without the support of many different people. First and foremost, I thank my advisor, Kevin Skadron, who has played an important role in nearly everything I have accomplished as a graduate student. Advisors have a tough job: they must walk a fine line between being too overbearing and being too laid back, with the right mix often depending heavily on the particular student. Kevin has always managed to find the right balance between the two extremes, consistently helping me stay confident and motivated but also challenging me to keep improving.

Other than my advisor, Nuwan Jayasena of AMD played the largest role in helping shape much of the work presented here. I also thank the members of my Ph.D. committee: Professors Jack Davidson, Sudhanva Gurumurthi, Mircea Stan, and Wes Weimer. Their guidance helped steer my work in a more interesting and useful direction. Wes deserves special recognition as a co-author of my first peer-reviewed publication, which was based on work that I completed as part of his Programming Languages course and which remains my most-cited (first-author) publication. Professor Andrew Grimshaw provided me with a unique opportunity to co-teach a course on parallel programming; this experience was rewarding and enlightening, and remains one of the highlights of my time in graduate school.

The members of the LAVA Lab have supported me in innumerable ways over the years. David Tarjan took me under his wing early in my career and was a wonderful collaborator on a number of different projects. Shuai Che, Jeremy Sheaffer, and Runjie Zhang provided feedback

on numerous matters both technical and otherwise. I am also indebted to other past and current members of the Lava Lab: Greg Faust, Marisabel Guevara, Tibor Horvath, Jiayuan Meng, Brett Meyer, Donnie Newell, Karthik Sankaranarayanan, Prateeksha Satyamoorthy, Lukasz Szafaryn, Jack Wadden, Ke Wang, and Liang Wang.

I am especially thankful for the support of my fellow graduate students Chris Gregg, Jiawei Huang, Sudeep Ghosh, and Nishant George throughout my time at the University of Virginia. Saurav Basu helped me understand the underlying algorithms and data structures used in the leukocyte tracking work. Jonathan Dorn provided valuable insight on the frequency scaling work. This dissertation was typeset using a LaTeX template created by Joel Coffman.

My parents encouraged my intellectual curiosity from a young age, and are in many ways responsible for my early interest in computers and my subsequent choice to study computer engineering as both an undergraduate and graduate student. Their support throughout the years has been unwavering despite the physical distance that has separated us for the last decade. Any homesickness I may have felt has been tempered slightly by my in-laws, Jay, Cathy, Alanna, and Brett, who have warmly welcomed me into their family here in Virginia.

Last but certainly not least, I thank my wife Jessica for her love and support. I am grateful for her constant encouragement, her occasional advice, and, perhaps most importantly, her patience. Seven years was a long time to wait; now it is time to see what adventures are in store for us in the next chapter of our life.

This work was supported in part by NSF grants IIS-0612049 and CNS-0916908, SRC grants 1607.001, 1972.001, and 2233.001, a GRC AMD/Mahboob Khan Ph.D. fellowship, an NVIDIA research grant, and equipment donated by AMD and NVIDIA.

Contents

Contents **vi**
List of Tables . . . viii
List of Figures . . . ix

1 Introduction **1**
1.1 Leukocyte Tracking . . . 3
1.2 Heterogeneous Load Balancing . . . 4
1.3 GPU Frequency Scaling . . . 5
1.4 Organization . . . 7

2 Background **8**
2.1 GPU Hardware . . . 8
2.2 GPU Software . . . 11
2.2.1 GPU Programming Model . . . 12
2.3 Energy Efficiency . . . 14

3 Case Study: Leukocyte Tracking **18**
3.1 Leukocyte Detection and Tracking . . . 20
3.2 Accelerating the Detection Stage . . . 22
3.2.1 Comparison to OpenMP . . . 23
3.2.2 Naïve CUDA Implementation . . . 24
3.2.3 CUDA Optimizations . . . 24
3.2.4 Single- vs. Double-Precision Performance . . . 26
3.3 Accelerating the Tracking Stage . . . 27
3.3.1 Comparison to OpenMP . . . 29
3.3.2 Naïve CUDA Implementation . . . 29
3.3.3 CUDA Optimizations . . . 29
3.4 Discussion . . . 35
3.4.1 Lessons for Software Developers . . . 36
3.4.2 Lessons for System Architects . . . 41
3.5 Related Work . . . 44
3.6 Conclusions . . . 44
3.7 Postscript . . . 45
3.8 Publications and Impact . . . 48

4 Heterogeneous Load Balancing **49**
4.1 Motivation 50
4.1.1 Need for Heterogeneous Load Balancing 51
4.1.2 Need for Dynamic Load Balancing 52
4.2 Related Work 54
4.3 Dynamic Load Balancing 57
4.3.1 Optimal Partition 57
4.3.2 Scheduling Algorithm 59
4.3.3 Scheduler Implementation 60
4.3.4 Example Schedule 61
4.4 Experimental Setup 63
4.4.1 Filtering Performance Anomalies 65
4.5 Results 67
4.5.1 Load Balancing without Variability 68
4.5.2 Load Balancing with Variability 72
4.5.3 Prediction Quality 78
4.5.4 Sensitivity to Chunk Size and Growth Rate 79
4.5.5 Sensitivity to Data Size 81
4.5.6 Severe Performance Imbalances 84
4.6 Conclusions and Future Work 84
4.7 Publications 86

5 GPU Frequency Scaling **87**
5.1 Background 89
5.1.1 CPU Frequency Scaling 89
5.1.2 GPU Frequency Scaling 92
5.2 Experimental Setup 94
5.2.1 Measuring Power Consumption and Execution Time 96
5.2.2 Controlling for Temperature 99
5.3 Reducing GPU Energy Consumption: Potential 100
5.4 Reducing GPU Energy Consumption: Algorithms 106
5.4.1 GreenGPU 108
5.4.2 Energy-Performance Tradeoff 111
5.4.3 ICE-AGE 112
5.4.4 Results 113
5.5 Reducing CPU Energy Consumption 118
5.6 Related Work 124
5.7 Conclusions and Future Work 125

6 Conclusions **129**

Bibliography **133**

List of Tables

2.1 Specifications of the top-of-the-line Intel CPU and AMD GPU as of March 2013. 10

3.1 Slowdown of the double-precision versions of the different CUDA implementations of the detection stage relative to the original, single-precision versions. 26

4.1 Average execution time and efficiency of Matrix Multiplication for native single-device execution and load balancing. 51
4.2 Load-balancing algorithm parameters. 60
4.3 Benchmarks and data sets used for evaluation. 64
4.4 Of 100 runs, number of runs discarded using a threshold of 2x. 67
4.5 Number of times each kernel must be run before the static approach can overcome its training overhead and reduce the total execution time relative the dynamic scheduler. 70

5.1 Benchmarks and data sets from the AMD APP SDK and Rodinia benchmark suite. 95
5.2 Core and memory clock frequencies used for frequency scaling. 100

List of Figures

2.1 CPU implementations of vector addition. 12
2.2 OpenCL kernel function implementing vector addition. 13

3.1 Still image from an intravital video of a mouse cremaster muscle. 21
3.2 Speedup of the different implementations of the detection stage over the original MATLAB implementation. 23
3.3 Speedup of the different implementations of the tracking stage over the original MATLAB implementation. 28
3.4 Impact of different overheads on the execution times of the CUDA implementations of the tracking stage. 30
3.5 Overall rates at which the four implementations can detect and track leukocytes. 35

4.1 Average native, single-device execution time of DCT as the core clock frequency of the discrete GPU is adjusted. 52
4.2 Optimal partition of work between the discrete and integrated GPUs for DCT over a range of discrete GPU clock frequencies. 53
4.3 Average execution time of DCT for three fixed partitions optimized for different frequencies, normalized to the best partition at each frequency. 54
4.4 Example schedule generated by the dynamic load balancer for the application DCT. 62
4.5 Optimal static schedule for DCT, discovered via an exhaustive search of all partitions. 62
4.6 Normalized throughput of kernel execution and data transfer for the integrated and discrete GPUs during 100 runs of dynamic load balancing of Black-Scholes. 66
4.7 Speedup of dynamic and static load balancing relative to single-device execution on the discrete GPU. 69
4.8 Performance loss of self-scheduling algorithms relative to our dynamic algorithm. 72
4.9 Average normalized execution time across a range of discrete GPU core frequencies for three fixed partitions and the dynamic scheduler, relative to the static oracle. 74
4.10 Average execution time of DCT for two fixed partitions and the dynamic scheduler, normalized to the best fixed partition at each frequency. 75
4.11 Maximum normalized execution time across a range of discrete GPU core frequencies for three fixed partitions and the dynamic scheduler, relative to the static oracle. 76

4.12 Performance loss of self-scheduling algorithms relative to our dynamic algorithm across a range of discrete GPU frequencies. 77
4.13 Speedup of the discrete GPU relative to the integrated GPU as a function of chunk size. 78
4.14 Time between the first and second devices finishing execution for the dynamic load balancer, as a fraction of the total execution time. 79
4.15 Average slowdown across all six applications relative to the optimal parameter values for each individual application. 80
4.16 Distribution of execution times for each application across the range of initial chunk sizes and chunk growth rates shown in Figure 4.15, normalized to the minimum observed execution time. 81
4.17 Speedup of dynamic load balancing relative to static load balancing for a range of data sizes. 82
4.18 Performance of the dynamic load balancer when one device is blocked, normalized to an oracle that sends all work to the unblocked device. 83

5.1 Impact of varying the core and memory frequencies on the execution time, power consumption, and energy consumption of Bitonic Sort. 101
5.2 Pearson's correlation between execution time and the two frequencies for each application. 103
5.3 Pearson's correlation between total power consumption and the two frequencies for each application. 104
5.4 The optimal compute energy savings due to DVFS. 105
5.5 Optimal clock frequencies for each application. 107
5.6 Increase in execution time versus decrease in compute energy for each optimal frequency pair. 112
5.7 Reduction in compute energy achieved by GreenGPU, Ice-Age, and the optimal, relative to running at the default frequencies. 115
5.8 Ratio of actual energy savings to optimal energy savings as a function of the performance cutoff C. 117
5.9 Impact of different waiting methods and maximum CPU frequencies on the execution time, power consumption, and energy consumption of Bitonic Sort and CFD running on the HD 5870. 120
5.10 Compute energy savings for two waiting methods and different maximum CPU frequencies, relative to busy waiting with a maximum frequency of 2.6 GHz. 123

Chapter 1

Introduction

For much of the history of computing, the steady growth in single-thread CPU performance has enabled consistent performance improvements without the need for substantial software changes. Both developers and end users could safely assume that existing software would execute significantly faster with each new hardware release. Over the past two decades, however, emerging thermal and power constraints forced an abrupt shift in the microprocessor industry. Rather than continuing to focus on maximizing the performance of a single thread running on a single core, designers began to create *multi-core* processors by integrating multiple independent cores into a single processor die. As a result, CPU core count has grown more rapidly than per-core performance, and legacy serial software no longer enjoys the same robust rate of performance growth. Exploiting concurrency has become essential for software to use multi-core processors to their full potential and continue to realize the benefits of semiconductor technology scaling.

One of the best examples of this hardware and software evolution is the *Graphics Processing Unit* or *GPU*. A GPU is a co-processor which the CPU can use to accelerate graphics rendering applications, such as three-dimensional video games. Traditionally, both the GPU's hardware design and software interface have been specialized for three-dimensional rendering, with limited utility beyond graphics. Demand for ever greater programmability, however, has led

GPUs to become general-purpose architectures, with full-featured instruction sets and rich memory hierarchies. Software tools such as NVIDIA's CUDA and the industry standard Open Compute Language (OpenCL) have made the process of developing general-purpose GPU applications accessible to a wider range of programmers.

Despite their increasingly general-purpose nature, GPUs are still designed primarily for efficient graphics rendering. Their design reflects the unique demands of graphics workloads: unlike many CPU applications, graphics applications are much more sensitive to aggregate throughput than they are to single-thread performance. As a consequence, GPUs are significantly more parallel than CPUs at a number of different architectural levels: they have more independent compute cores, more parallelism within each core, and more connections between the cores and memory. This massive parallelism provides GPUs with a potentially enormous performance advantage: the fastest available GPU can provide approximately six times the memory bandwidth and twenty times the computational throughput of the fastest available CPU.

In addition to their tremendous performance potential, GPUs are also ubiquitous: all laptop and desktop computers and many server systems include a GPU, either integrated into the motherboard or CPU or installed in an expansion slot as a discrete device. Some consumer systems even include two or more GPUs: in a desktop, multiple discrete GPUs can execute concurrently to improve performance, while in a laptop, separate integrated and discrete GPUs can be selected at run time to provide a tradeoff between power consumption and performance. Supercomputers are also increasingly being provisioned with large numbers of GPUs: Titan, the fastest supercomputer in the world as of November 2012, contains more than 18,000 NVIDIA Tesla K20X GPUs [102]. At the other end of the performance spectrum, smart phones are being outfitted with increasingly powerful GPUs as well, as companies try to keep pace with the growing demands of graphics-intensive mobile computing applications.

GPU-enabled systems represent one of the most interesting and widespread examples of *heterogeneous systems*, in which computational devices of different capabilities are combined

into a single system. The performance gap between the different devices in such systems may be large, spanning an order of magnitude or more for many applications. Optimizing applications for systems with such extreme levels of performance heterogeneity is quite challenging. Although GPU-enabled heterogeneous systems offer the potential for high performance and low energy consumption, in practice many applications that leverage them are wasteful of both performance and energy. This wastefulness is typically a byproduct of the underutilization of resources.

In this dissertation, we propose and evaluate techniques for addressing the underutilization of resources in three different scenarios that commonly arise in heterogeneous CPU-GPU systems. First, we observe that straightforward implementations of iterative algorithms often result in significant underutilization of the GPU because interacting with the GPU involves relatively severe software overheads. We present general techniques for reducing these overheads and improving GPU utilization and performance. Second, we observe that most GPU applications only utilize one device, even in systems with multiple powerful devices. We propose a novel, dynamic scheduling algorithm that automatically load balances an arbitrary GPU kernel across multiple devices, keeping each device fully utilized and improving the overall performance. Third, we observe that most GPU kernels only fully utilize either the compute cores or the memory system. We propose a novel DVFS algorithm for GPUs that slows down the underutilized resource to increase its effective utilization and improve the overall energy efficiency.

1.1 Leukocyte Tracking

To better understand the challenges and tradeoffs associated with implementing an iterative algorithm on a GPU, we use as a case study a computationally-intensive video processing application from systems biology: the tracking of leukocytes or white blood cells. This application is of significant interest to medical researchers but suffers from performance that

is far from the desired real-time speed. In this dissertation, we describe in detail our efforts to dramatically accelerate leukocyte tracking using a GPU-enabled system and make the following general contributions:

- We highlight performance bottlenecks that reduce GPU utilization, most of them related to the high cost of CPU-GPU interaction. These bottlenecks are likely to limit the performance of iterative applications that may be ported to the GPU in the future.

- We discuss in detail algorithmic and programmatic transformations that can be used by programmers to bypass or ameliorate these bottlenecks. These transformations are general enough to be applied to a wide range of applications, potentially allowing such applications to obtain a greater performance benefit from the GPU.

- We provide concrete suggestions for ways in which system architects can reduce the negative impact of these bottlenecks through both software and hardware modifications. The implementation of our suggestions would allow programmers to leverage the GPU more efficiently with less effort.

1.2 Heterogeneous Load Balancing

In a GPU-enabled system, applications requiring high performance will typically target only the fastest single device during computationally-intense phases. This approach is wasteful in a system with a powerful multi-core CPU or multiple GPUs, because it leaves the other devices in the system idle and unutilized. Unfortunately, developing an application that can effectively utilize all available devices, and do so consistently across a wide range of diverse systems, is challenging because a device's execution rate, and thus the optimal work distribution, is highly application dependent and may vary significantly at run time.

Prior work has proposed load-balancing frameworks that divide work across multiple devices with minimal programmer effort [50, 61]. However, these frameworks either assume

that every device's execution rate is equal [50] (i.e., that the hardware is homogeneous) or that a device's execution rate does not vary [61]. To overcome these limitations, in this dissertation we explore dynamic load-balancing techniques and make the following contributions:

- We present a novel dynamic scheduling algorithm for heterogeneous systems that can load-balance applications automatically with no offline training and can respond effectively to variations in execution rates at run time.
- We show that our scheduling algorithm nearly matches the performance of the best possible static partition when device-level performance is fixed and provides better performance when device-level performance varies.
- We also show that our scheduling algorithm can provide resilience in the face of severe performance imbalances caused by hardware or software failures or starvation.

1.3 GPU Frequency Scaling

One popular approach for reducing CPU power consumption is based on the observation that the performance of many applications is limited by the memory system and not the CPU itself. In such applications, the CPU is underutilized, and we can safely reduce the CPU's voltage and clock frequency to decrease power consumption without hurting performance. This technique is known as dynamic voltage and frequency scaling (DVFS) [106]. The use of DVFS is predicated on the existence of the necessary hardware support, but typically the decision about how and when to adjust the clock frequency is delegated to software.[1]

GPUs present a ripe target for DVFS for two primary reasons. First, the maximum power consumption of a typical high-end GPU is approximately twice that of a typical high-end CPU. In many cases, the power consumption of the GPU, especially under load, can represent the majority of a system's overall power consumption. Second, unlike CPUs, GPUs typically

[1] Although DVFS varies both the voltage and the clock frequency, typically it is only the frequency changes that are made visible to software.

allow software to adjust the frequency of both the compute and memory systems. Applying DVFS in this context is more complicated, because we must first identify which resource in the GPU is underutilized (the compute or memory system) and then decrease its frequency to reduce power consumption without impacting performance.

Recent work has extended an existing CPU DVFS algorithm [31] to GPUs and demonstrated substantial energy savings using a GPU that was released seven years ago [62]. Unfortunately, the proposed algorithm performs poorly on modern GPUs from a different vendor, resulting in higher energy consumption than simply leaving the GPU's clock frequencies at their default levels. In this dissertation, we explore novel techniques for reducing the energy consumption of GPUs using DVFS and make the following contributions:

- We demonstrate that dynamic voltage and frequency scaling (DVFS) can provide significant energy savings for modern GPUs, using actual power measurements from a real system.

- Although prior work has shown that, for CPUS, we must make a tradeoff between high performance and low energy CPUs [27, 31], and the only existing GPU DVFS algorithm is explicitly designed based on this assumption [62], we show that maximizing performance and minimizing energy are *not* conflicting goals for GPUs.

- Based on this key observation, we propose a simple heuristic for predicting the energy-optimal clock frequencies for an application based only on easily measurable performance data. Because our heuristic does not require power measurements, it can be applied immediately to existing commodity systems without the need for special-purpose hardware.

- We show that our heuristic works well across multiple generations of GPUs, providing energy savings within 87% of the optimal on average.

- We demonstrate that the CPU also wastes a significant amount of energy during GPU execution; we present two techniques for addressing this problem and show that they can provide significant energy savings.

1.4 Organization

The rest of this dissertation is organized as follows: Chapter 2 provides background information on the evolution of GPU computing and the increasing importance of energy efficiency, Chapter 3 discusses our experiences accelerating leukocyte tracking, Chapter 4 explores heterogeneous load balancing, Chapter 5 presents techniques for applying DVFS to GPUs, and Chapter 6 concludes.

Chapter 2

Background

In this chapter, we discuss three important trends during the past decade: improvements in the performance and programmability of GPU hardware, the maturation of GPU software, and the increasing importance of energy efficiency.

2.1 GPU Hardware

In early 2003, the most advanced GPU and CPU from NVIDIA and Intel, respectively, offered approximately the same theoretical peak single-precision performance. Nine years later, in 2011, the peak performance of NVIDIA's most advanced GPU was *eight* times that of Intel's most advanced CPU. GPUs have been able to provide such rapid performance growth relative to CPUs because GPU workloads are more sensitive to aggregate throughput than single-thread performance. GPU designers have taken advantage of this fact by replicating simple processing elements (PEs) rather than using large, expensive cores, and by devoting much less die area to caches and control logic. Groups of the simple PEs are harnessed together under SIMD (Single Instruction Multiple Data) control, amortizing the area and power overhead of the instruction store and control logic. Instead of large caches, GPUs cope with memory latency with a combination of massive multithreading (thousands of hardware thread contexts) and higher memory bandwidth. Even if some threads are stalled waiting

for data from memory, there will likely be many other threads that can be executed in their place. This ensures high utilization and high throughput, but at the expense of increased latency for any particular thread.

These massive increases in performance have been coupled with increases in the programmability and flexibility of GPUs [78]. In an effort to allow graphics programs to generate increasingly realistic images, GPU architectures have evolved from a rigid set of fixed-function pipeline stages into a fully programmable pipeline. Changes include support for more complex control flow instructions and less rigid restrictions on memory access patterns. In more recent years, GPUs have also added features specifically for the benefit of non-graphics, general-purpose computation, such as support for higher-precision floating-point arithmetic and ECC memory, both critical requirements for many applications, and much lower overhead global synchronization, which is useful for iterative computations.

As a concrete example of the differences between CPUs and GPUs, Table 2.1 compares the best available Intel CPU and AMD GPU using a number of different metrics. The GPU's massive parallelism is clearly evident in many of these metrics. While the CPU has 10 independent compute cores, the GPU has 32 independent compute units (the rough analog of a CPU core). A single CPU core has 2 hardware thread contexts, while a single GPU core has 2,560 thread contexts, up to 64 of which may be active in a single clock cycle. The GPU can thus have 2,048 threads active concurrently, more than 100 times as many as the CPU. The difference in peak performance is not quite as high, however, because the CPU runs at a much higher clock frequency and can perform more computations per thread per cycle. Overall, the GPU's peak computational throughput is 21 times higher than that of the CPU. The GPU has a smaller but still significant advantage when it comes to memory bandwidth. Because the GPU has three times as many memory channels, significantly wider memory interfaces, and higher memory clock speeds, it achieves a 6.7 times higher peak memory bandwidth. It also consumes significantly more power than the CPU, although the

	Intel CPU	**AMD GPU**	**GPU/CPU Ratio**
Die Size (mm^2)	513	352	0.7
Cores	10	32	3.2
Thread contexts per core	2	2,560	1280
Total thread contexts	20	81,920	4096
Active threads per core	2	64	32
Total active threads	20	2,048	102
Core Frequency (GHz)	2.4	1.0	0.4
Peak throughput (GFLOPS/s)	192	4,096	21
Total cache (MB)	33.1	1.4	0.04
Memory channels	4	12	3
Peak memory bandwidth (GB/s)	43	288	6.7
Peak power consumption (W)	130	250	1.9
Release price	$4,616	$499	0.11

Table 2.1: Specifications of the top-of-the-line Intel CPU and AMD GPU as of March 2013: the Intel Xeon E7-8870 [28, 29] and the AMD Radeon HD 7970 GHz Edition [6–8].

increase in peak compute and memory throughput is much higher than the increase in power consumption.

An emerging trend over the past few years has been the combination of a CPU and a GPU onto a single die. Both AMD and Intel released such processors in 2011, under the brand names AMD Fusion[1] and Intel Sandy Bridge. The CPU and GPU in these fused designs share a single memory system, unlike the separate memories employed in traditional discrete designs. Thus, one potential benefit is a much lower or even negligible overhead in communicating data between the CPU and GPU. However, the major limitation of all existing fused offerings is that they include relatively weak GPUs, in terms of both compute throughput and memory bandwidth. This points to a potential limitation of fused designs, one based primarily on financial rather than technological considerations: it is unlikely that AMD will release a Fusion chip incorporating a high-performance GPU, because it would risk cannibalizing sales of its high-end discrete GPUs. Intel might conceivably be free from such economic constraints, but it does not produce high-performance GPUs. For certain

[1]For legal reasons, AMD now markets these processors as Accelerated Processing Units (APUs) instead of Fusion processors.

applications, the reduced data transfer overhead outweighs the longer computation time and a Fusion GPU can thus provide a speedup over a more powerful discrete GPU [30]. However, discrete GPUs will likely remain the optimal choice for most applications requiring high performance.

2.2 GPU Software

The tremendous growth in GPU performance and flexibility has led to an increased interest in performing general-purpose computation on GPUs (GPGPU) [78]. Early GPGPU programmers wrote programs using graphics APIs. This had the benefit of exposing powerful graphics-specific hardware, but incurred the programming and execution overhead of mapping a non-graphics computation onto the graphics API and execution stack. Recognizing the burgeoning interest in GPGPU, the two largest discrete GPU vendors have released a series of software tools designed to simplify the development of GPGPU applications. In 2006, AMD released Close-to-the-Metal (CTM) [91], which, as its name implies, is a low-level interface for GPU programming that bypasses the graphics API. NVIDIA took a higher-level approach with its tool, CUDA[2] [71], released in 2007. CUDA defines an API for interacting with the GPU and a C-based programming language for expressing GPU computations.

More recently, the Open Computing Language (OpenCL) has emerged as an industry standard programming language for heterogeneous computing. Development of the OpenCL specification was initiated by Apple but is now managed by the Khronos Group, an industry consortium. The first version of the specification was released in late 2008, with both AMD and NVIDIA releasing compliant implementations in 2009. AMD has since dropped support for CTM and thrown its full weight behind OpenCL. A number of other companies, including Apple, ARM, IBM, Intel have also announced support for OpenCL. Because of this industry-wide support, OpenCL kernels can run on two important classes of hardware that CUDA

[2] Officially, CUDA is no longer an acronym, although originally it stood for "Compute Unified Device Architecture".

```
1 for (int i = 0; i < N; i++) {
2     C[i] = A[i] + B[i];
3 }
```

(a) Serial implementation.

```
1 #pragma omp parallel for
2 for (int i = 0; i < N; i++) {
3     C[i] = A[i] + B[i];
4 }
```

(b) OpenMP implementation.

Figure 2.1: CPU implementations of vector addition.

kernels cannot: CPUs and non-NVIDIA GPUs. The first category includes CPUs from AMD, ARM, IBM, and Intel; the second includes GPUs from AMD, ARM, and Intel.

2.2.1 GPU Programming Model

CUDA and OpenCL utilize extremely similar programming models. Although we use OpenCL terminology in the following discussion, CUDA provides equivalent functionality except where noted otherwise.

The OpenCL system model consists of a *host* device (a CPU) for running serial portions of an application and one or more *compute* devices for running parallel sections. Note that the two categories are not mutually exclusive[3]; the host CPU can also act as a compute device. The host interacts with a device through the OpenCL API, which allows the host to request the allocation of resources, transfer data between host and device memory, and schedule commands for execution. Data-parallel computations are expressed as *kernel* functions that get executed at every point in an application-defined *domain*, which can be one-, two-, or three-dimensional. A single point in the domain is called a *work item*, and the domain is sub-divided into groups of neighboring work items called *work groups*.

As a concrete example, consider how we might parallelize the computation of the vector sum $C = A + B$. Figure 2.1a shows a straightforward serial implementation that uses a `for` loop to iteratively compute the value of each element of C. Although there are many possible ways that we could parallelize this computation on a CPU, the simplest approach would

[3]In CUDA, the categories *are* mutually exclusive: the only compute device currently supported is an NVIDIA GPU.

```
1 __kernel void add_vectors(float *A, float *B, float *C, int N) {
2     int i = get_global_id(0);
3     if (i < N) {
4         C[i] = A[i] + B[i];
5     }
6 }
```

Figure 2.2: OpenCL kernel function implementing vector addition.

be to use a high-level programming model like OpenMP [77]. Figure 2.1b shows a basic OpenMP implementation that parallelizes the `for` loop using a special `pragma` statement, which instructs the OpenMP runtime system to evenly divide the iterations of the `for` loop across the available CPU cores or thread contexts.

A common way to transform a serial implementation like this one into a parallel kernel is to use, with some minor modifications, the body of the `for` loop as the body of the kernel function. Figure 2.2 shows the OpenCL kernel[4] that results from transforming the serial vector addition implementation (from Figure 2.1a) in this manner. Recall that the kernel body encodes the behavior of each work item. In line two, each work item first determines its location i in the domain (in this case, a one-dimensional domain) using a built-in OpenCL function. If this location corresponds to a valid vector element (i.e., it is not beyond the end of the vector), then the work item computes a single value of the result vector in line four. Note that this is certainly not the only possible way to implement vector addition as a kernel function. For example, each work item could compute the values of multiple elements in the result vector, rather than only computing a single element.

How a kernel gets mapped onto hardware depends on the type of processor being used. On a CPU, a work group gets mapped onto a single CPU thread; the thread iterates through the set of work items, executing them sequentially. Parallelism is achieved by mapping different work groups onto different CPU cores. On a GPU, a work group gets mapped onto a SIMD

[4]For simplicity, we only show the kernel function here. We omit the corresponding host code which would be responsible for transferring the input vectors to the GPU, invoking the kernel (with the correct number of work items), and transferring the output vector back to the CPU.

core, which executes multiple work items concurrently. Parallelism is thus achieved both within a single work group and across multiple work groups that get mapped onto different GPU cores.

A group of work items that executes together is called a *wavefront* on AMD GPUs and a *warp* on NVIDIA GPUs and consists of 64 or 32 work items, respectively.[5] Although it is possible to write a correct kernel without taking this information into account, in many cases it is impossible to achieve high performance without doing so. In particular, control-flow divergence or memory divergence (work items accessing non-contiguous memory locations) within a wavefront or warp can have severe performance consequences.

The size of the work group is set by the programmer, but an upper limit is determined by the compute device being used; work groups on the latest AMD and NVIDIA GPUs, for example, can contain no more than 256 and 1,024 work items, respectively. Synchronization and communication are only allowed within a work group[6], not among different work groups, and work groups may be executed in any order. Together these two properties, limited work group size and independence of work groups, help ensure scalability by setting an upper limit on the number of work items that might need to communicate.

2.3 Energy Efficiency

Energy has long been a first-class concern in portable, battery-powered systems. It is only in more recent years that the energy consumption of desktop and server systems has become an important consideration. Although performance is still king in many domains, energy is increasingly considered a hard design constraint if not a metric to be optimized. When we optimize solely for energy, we are attempting to maximize *energy efficiency*, often expressed as the number of operations per joule. We can alternatively jointly optimize for performance

[5] Note that a warp or wavefront is not the same as a work group: the former is a byproduct of the way that GPU hardware is implemented while the latter is an explicit software construct.

[6] There is one exception: atomic operations can be used for global synchronization, but only when the number of work groups is small enough that all are simultaneously active, which is typically not the case.

and energy by trying to minimize metrics such as the energy delay product (ED) or energy delay squared (ED^2). Note that optimizing for energy is not the same as optimizing for power; techniques that decrease power often also reduce performance and can thus lead to an overall increase in energy. For example, decreasing power consumption by 2x at the cost of a 3x increase in total execution time leads to an overall $\frac{3}{2} = 1.5$x increase in energy.

Energy is of particular concern in large-scale systems; for example, the majority of the total cost of ownership of a supercomputer is due to the cost of purchasing electricity to power the system [87]. Los Alamos National Laboratory recently announced that Roadrunner, the fastest supercomputer in the world as recently as June 2009 [101] and still one of the 25 fastest supercomputers as of November 2012 [102], has been decommissioned because its energy efficiency was insufficient "to make the power bill affordable" [60]. The power consumption of computing equipment is not the only concern; approximately half of the energy used by a data center is consumed by other components, such as the power delivery and cooling systems [20]. Reducing the power consumption of the computing equipment can have a commensurate impact on the power consumption of these other components. As the number of data centers as well as their individual energy consumptions both continue to grow, data centers represent an ever-larger fraction of total global energy usage: data center power usage doubled from 2000 to 2006 and already represented 1.5% of the total electricity usage in the United States as of 2006 [20].

One important step on the road to enabling improvements in energy efficiency has been the development of industry standard power management interfaces, of which the Advanced Configuration and Power Interface (ACPI) [2] is the best example. ACPI provides a mechanism by which software (typically the operating system) can make tradeoffs between the performance and power consumption of the various components in a system. ACPI defines different performance states (P-states), which reduce power at the expense of lower performance, and different sleep states (C-states), which reduce power at the expense of higher latency before a device is returned to an active state. The mere existence of these

low power states does not guarantee energy efficiency, however; software must decide how to leverage the various states judiciously in order to reduce energy while maintaining acceptable performance.

Related to energy efficiency is the concept of energy proportionality, the notion that the amount of energy consumed by a system should be proportional to the amount of work it completes [10]. Although the vast majority of computers are most energy efficient at higher utilization, they unfortunately spend most of their time underutilized and thus in a less energy-efficient state. Note that proportionality and energy efficiency are related but distinct concepts: proportionality simply means that a system's energy usage is proportional to its utilization but says nothing about its energy efficiency at any particular utilization. We might argue that, for systems with varying utilization, energy proportionality is a necessary but not sufficient condition for energy efficiency.

The two sources of power consumption in integrated circuits are dynamic power and static power. Dynamic power is due to transistors switching on and off and is therefore directly related to the utilization (or activity factor) of a circuit. Static power, on the other hand, is primarily due to undesired leakage current which flows even when a transistor is nominally off. Leakage current has increased substantially as process technologies have shrunk, and is also exponentially dependent on temperature [47]. This temperature dependence can result in a positive feedback loop, because temperature is also dependent on power consumption. Decreasing temperature, in addition to decreasing leakage power, decreases cooling costs and often improves reliability.

Another source of wasted power consumption in a computer system is the power supply, which is responsible for converting the high-voltage AC input signal into multiple low-voltage DC output signals to power the various components in the system. This conversion process is not perfectly efficient and results in wasted power. The efficiency of a power supply is typically an increasing function of its load, except at loads near the power supply's maximum

rated load, where efficiency typically decreases. A decent modern power supply can achieve efficiencies of at least 80% at loads ranging from 10% to 100% of its maximum rated load [34].

We discussed earlier the performance implications of leveraging GPUs; a GPU can also provide substantial energy savings. This is because a high-end GPU can provide more than an order of magnitude higher peak compute performance than a high-end CPU while only dissipating around twice as much power. Thus, GPUs may be attractive regardless of whether our goal is to maximize performance or minimize energy.

Chapter 3

Case Study: Leukocyte Tracking

This chapter discusses our experiences parallelizing a computationally intensive application from the field of systems biology: the detection and tracking of rolling leukocytes (white blood cells) in *in vivo* video microscopy of blood vessels [33, 81]. Tracking leukocytes provides researchers with important information about the inflammatory response of the vascular system. Unfortunately, manual tracking is a tedious process, requiring on the order of tens of hours of manual analysis for video from a single experiment [81]. Automated approaches to the detection [33] and tracking [81] of leukocytes obviate the need for manual analysis, but are computationally expensive, requiring more than four and a half hours to process one minute of video on state-of-the-art hardware. Significantly reducing the execution time of these automated approaches would help accelerate the process of developing anti-inflammatory medications.

This particular application was targeted for acceleration partially because it demonstrates an urgent need for dramatic speedups. More importantly, however, it is a useful case study because it illustrates many of the issues that other applications will face in trying to leverage many-core systems. It is a complex application that presents nontrivial software engineering challenges and is representative of a much broader class of applications. The application's execution time is dominated by a number of widely used operations, such as stencil-based

operations (specifically feature extraction and image dilation) and an iterative solution procedure. Both operations are widely used in image processing and the last operation is widely used in high performance computing.

The detection and tracking algorithm was originally implemented in MATLAB [33, 81]. We first reimplemented the entire application in C, which resulted in a significant performance improvement. We further improved the performance by accelerating the most computationally demanding stages using CUDA[1] and, for comparison, OpenMP. Relative to the original MATLAB implementation, we achieved an overall speedup of 200x using a desktop system with an NVIDIA GeForce GTX 280 GPU, compared to a speedup of 7.6x on the fastest available multi-core CPU system. These speedups demonstrate the advantages of the throughput-oriented nature of GPUs. Additionally, we have identified a number of bottlenecks, in both hardware and software, whose elimination would enable even more significant speedups and simplify the development of efficient CUDA applications.

In addition to the substantial speedup, the main contribution of this chapter is to describe in detail the algorithmic transformations needed to reduce the overheads associated with a separate coprocessor. These overheads are particularly acute with iterative algorithms such as iterative solvers. The best implementation required abandoning the canonical parallelization strategy suggested in the CUDA literature, in which each output value is computed by a separate thread. We also propose changes to CUDA's software and hardware architecture that would provide better support for applications with fine-grained interleaving of serial and parallel regions.

The work presented in this chapter was completed in early 2009. Some aspects of GPU hardware and software have evolved significantly since that time, and some of the observations we make here have become less relevant with time. This is discussed in greater detail in Section 3.7.

[1] OpenCL did not yet exist at the time this project was initiated. We have since ported the application to OpenCL as well.

3.1 Leukocyte Detection and Tracking

Leukocytes, or white blood cells[2], play an important role inside the body, acting as the body's defense mechanism against infection and cellular injury. Much effort has been invested in studying the way leukocytes carry out this role in the process of inflammation. The most commonly used statistic predicting the level of cell recruitment during inflammation is the velocity distribution of rolling leukocytes [36, 82]. Knowing this distribution can aid researchers in understanding the mechanisms behind leukocyte rolling and arrest in order to create effective inflammation treatments. As a result, researchers investigating anti-inflammatory drugs need a fast, accurate method of attaining these measurements to test the validity of their treatments.

Currently, the velocity distribution is measured manually. Researchers go through hours of video data frame-by-frame, marking the centers of rolling leukocytes at an average rate of several minutes per leukocyte [3, 36, 82]. To obtain a valid estimate of the leukocyte velocity distribution, hundreds of cells must be tracked. This process requires many tiresome hours and, like any human action, involves a certain amount of observer bias. Automatically tracking cells addresses both of these problems and allow researchers to focus more on the problem of creating treatments and less on the tabulation of data. Furthermore, the possibility of real-time leukocyte detection and tracking would open new avenues for experimentation by allowing a researcher to vary experimental parameters until appropriate results are obtained, rather than conducting many different experiments separated by periods of data tabulation and analysis.

Automatic detection is accomplished using a statistic called the Gradient Inverse Coefficient of Variation (GICOV) [33]. The GICOV measures the mean outward gradient magnitude along a closed contour divided by the standard deviation of the same. In the implementation used here, the contours are restricted to circles of a known range of radii. In the first image of a sequence, detection is performed on the whole image, because leukocytes may be present at

[2] We use the terms *leukocyte* and *cell* interchangeably in this chapter

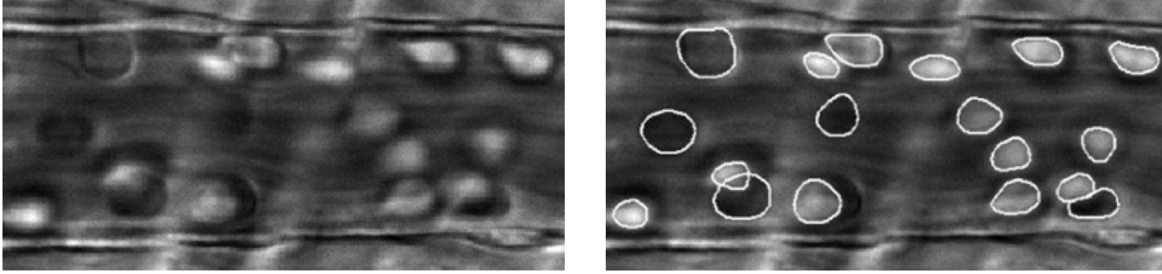


Figure 3.1: Still image from an intravital video of a mouse cremaster muscle. On the left is the original image; on the right is the result of automatic detection with the leukocytes outlined.

any location in the frame. Following the initial detection, subsequent detections only need to be performed in a small window at the entry side of the venule in order to detect leukocytes entering the field of view.

After detection, an active contour (snake) algorithm is used to track the boundary of each leukocyte from frame to frame using a statistic called the Motion Gradient Vector Flow (MGVF) [81]. The MGVF is a gradient field biased in the assumed direction of the movement of the leukocytes (i.e., the direction of blood flow). This active contour method works well in the cluttered, contrast-varying scene encountered in intravital microscopy. The snake is tailored to the leukocyte model and is constrained to prefer circular shapes of a radius near the average radius for leukocytes of a given species.

The images used for detection and tracking are of leukocytes found in vivo, that is, within a living organism. The video used in this chapter was made using intravital microscopy, filming the cremaster muscle of a mouse. This muscle is particularly thin, making it transparent, and is filled with post capillary venules. Part of a frame from such a movie is shown in Figure 3.1. These intravital images present a salient challenge for automated image analysis.

The particular video used in this work is a 640 x 480 uncompressed AVI file. The actual blood vessel being analyzed only occupies approximately a third of each frame, so a frame is cropped to a 218 x 480 sub-frame before detection is performed. The cropping boundary is hard-coded for all performance measurements, although in practice it would be designated manually by the user in the first frame. During the tracking stage, only a small, fixed-sized

area around each cell is analyzed, so the performance of the tracking stage is a function of the number of cells being tracked rather than the size of the frame. The video was recorded at 30 frames per second (FPS), so achieving real-time analysis would require processing each frame in 1/30th of a second.

3.2 Accelerating the Detection Stage

In order to automatically detect leukocytes in an image, three operations are performed. First, for each pixel in the image, the GICOV score is computed for circles of varying radii (stencils) centered on that pixel, and the maximum of those scores is output into a matrix. Second, this matrix is dilated[3], which simplifies the process of determining if the GICOV score at a given pixel is the maximum within that pixel's neighborhood. Third, for those pixels which have locally maximum GICOV scores, an active contour is used to refine the initial circle and more precisely determine both the location and the shape of the leukocyte.

Previous work implemented both the detection [33] and tracking [81] stages of the algorithm using MATLAB. In the detection stage of that implementation, the GICOV computation and dilation take 36.7% and 28.2% of the overall execution time, respectively. In our C implementation of the algorithm, these two operations further dominate the execution, taking 59.1% and 39.2% of the execution time, respectively. The C implementation is essentially a line-by-line translation of the MATLAB implementation and provides a speedup of 2.0x on the detection stage. To further improve the performance of the C implementation, we accelerated the critical operations using OpenMP and CUDA. The OpenMP acceleration was achieved with the introduction of two simple pragmas. The CUDA acceleration was more complex, starting with a straightforward translation and then applying increasingly complex optimizations. The speedups achieved by the C, OpenMP, and CUDA implementations of the detection stage are shown in Figure 3.2.

[3]Dilation takes as input an $M \times N$ matrix and a stencil or neighborhood, and outputs an $M \times N$ matrix in each which each element i is the maximum of all of the values in the neighborhood of i in the input matrix.

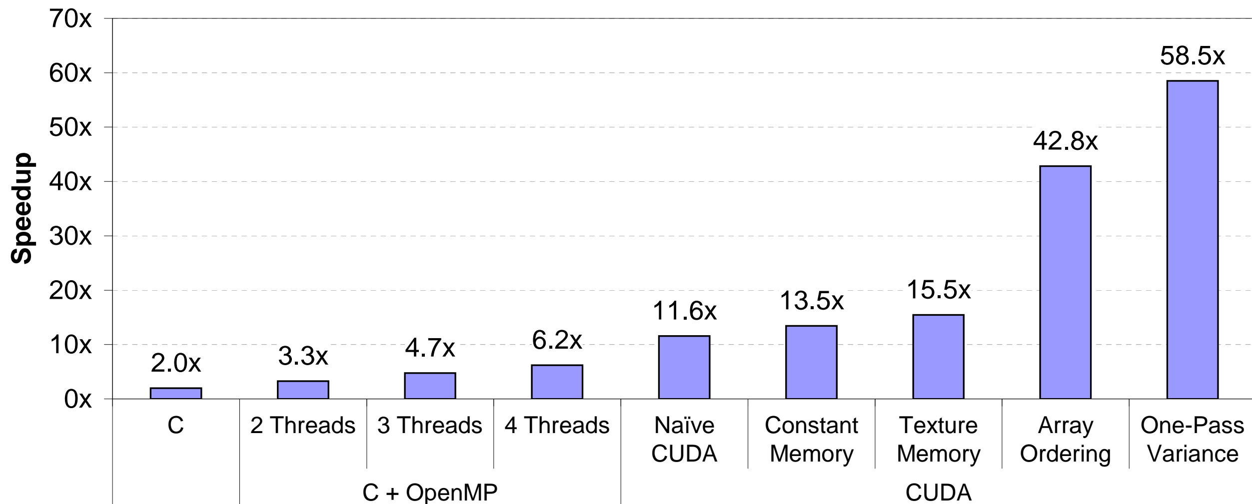


Figure 3.2: Speedup of the different implementations of the detection stage over the original MATLAB implementation.

Execution times for all implementations were measured on a machine running Ubuntu version 7.10 with a 3.2 GHz quad-core Intel Core 2 Extreme X9770 processor and an NVIDIA GeForce GTX 280 GPU, with NVIDIA driver version 177.67 and CUDA 2.0. The original implementation was executed in MATLAB version 7.6.0.324. The C code was compiled using GCC version 4.2.3 and the CUDA code was compiled using NVCC version 0.2.1221. The first access to the CUDA API incurs a non-negligible delay due to initialization overhead. Because a real-time implementation would initialize the API before the video capturing begins, and because this delay can vary significantly between different runs, we started measuring the execution time after a dummy call to the API.

3.2.1 Comparison to OpenMP

A popular approach to parallelizing programs on shared-memory machines is the OpenMP standard [77]. In order to provide a point of comparison to our CUDA implementations, we have also parallelized the leukocyte detection using OpenMP. Specifically, the `for` loops in the GICOV computation and dilation functions that iterate over the pixels in the image were augmented with a `parallel` pragma. This transformation was trivial because the `for` loops contain no inter-loop dependencies. The OpenMP speedups for two, three, and four threads

are shown in Figure 3.2. With four threads, the OpenMP implementation achieves speedups of 6.2x and 3.1x over the original MATLAB and C implementations, respectively.

3.2.2 Naïve CUDA Implementation

The CUDA implementations parallelize exactly the same loops as in the OpenMP approach. The code inside the nested `for` loops in each function was converted directly into a kernel function, and the domains of the kernel functions were defined to be the pixels in the image; in other words, each CUDA thread was assigned the computation for a single pixel. This straightforward CUDA implementation achieves a 5.9x speedup over the original C version.

3.2.3 CUDA Optimizations

Although the naïve CUDA implementation achieves a non-trivial speedup over the sequential version of the application, it was written without taking into account the unique architecture of the GPU. A number of optimizations were applied to the original CUDA implementation that significantly improved its performance. Each optimization is described in turn. Note that the optimizations are cumulative, meaning that once an optimization has been applied, it remains in effect in all subsequent optimizations. However, they are independent of each other and could be applied in any order. For each optimization, we also note its applicability to programs parallelized using OpenMP.

Constant Memory: Many of the arrays accessed by both kernels are read-only and relatively small. Thus, they can be allocated in the GPU's special-purpose constant memory address space, which allows them to be cached on-chip. Accomplishing this change in CUDA is trivial, but it allows the code to achieve a speedup of 6.8x over the original C version and 1.16x over the naïve CUDA version. This optimization is not applicable to the OpenMP version, since CPU architectures do not provide such special purpose address spaces.

Texture Memory: GPUs also employ another special-purpose address space for texture memory. Like constant memory, texture memory only supports read-only data structures.

Data structures mapped to texture memory can take advantage of special hardware on the GPU which provides a small amount of on-chip caching and slightly more efficient access to off-chip memory. By moving the large arrays accessed by the two kernels into texture memory, the application achieves a speedup of 7.8x over the original C version and 1.15x over the previous CUDA version. As texture memory is an architectural feature of GPUs, this optimization is not applicable to the OpenMP version.

Array Ordering: The two largest arrays accessed by the GICOV kernel were originally allocated in row-major order. The memory access pattern of the kernel resulted in threads within the same warp accessing non-contiguous elements of the arrays. Allocating the arrays in column-major order allows threads within the same warp to access contiguous elements, which can significantly improve performance due to the GPU's ability to coalesce multiple contiguous memory accesses into one larger memory access. With this optimization, the detection stage achieves a speedup of 21.7x over the original C version and 2.8x over the previous CUDA version. Programs using OpenMP on many-core CPUs such as Sun Niagara 2 [69] and Intel Larrabee [92] can benefit from this optimization, as it makes more efficient use of the L1 data cache and memory bandwidth when working on many data points in parallel. It does not impact the running time of our OpenMP implementation, however, as each heavyweight core processes a single data point at a time and can fully buffer the array in the L1 cache in both layouts.

One-Pass Variance: For each point in the image, the GICOV kernel computes the sum of a function at 150 different points and then computes the variance of the function across those same points. This two-pass approach is inefficient because it requires storing the 150 intermediate values, which requires spilling those values to global memory[4]. The variance can instead be computed in a single pass using a relatively straightforward algorithm [107]. This optimization provides an overall speedup of 29.7x over the original C version and 1.37x over the previous CUDA version. We experimented with this optimization in the OpenMP

[4] Another option would be to spill to the on-chip shared memory. Because the shared memory is small, however, this approach reduces the number of threads per core and reduces overall performance.

Implementation	Slowdown
Naïve CUDA	1.78x
Constant Memory	1.96x
Array Ordering	1.49x
One-Pass Variance	1.88x

Table 3.1: Slowdown of the double-precision versions of the different CUDA implementations of the detection stage relative to the original, single-precision versions.

version and observed no speedup, since the L1 cache in each CPU core is large enough to buffer the 150 intermediate values.

3.2.4 Single- vs. Double-Precision Performance

Early releases of CUDA only supported single-precision floating point arithmetic due to the limitations of the underlying hardware. Recent releases of CUDA, however, fully support double-precision arithmetic as a result of the hardware support offered in NVIDIA's most recent GPUs, such as the GeForce GTX 280 used in this work. There is an important caveat, however: although each core in the GTX 280 contains its own single-precision floating point ALU, the double-precision floating point ALUs are actually shared among 8 SMs. Thus, for compute-bound kernels, switching from single- to double-precision should reduce performance by approximately a factor of eight.

The original MATLAB and C implementations use double-precision values throughout the application. For the results presented earlier, the CUDA implementations use single-precision values in the two GPU kernels but retain the use of double-precision values in the rest of the computation. Four of the CUDA implementations[5] described above were also reimplemented using double-precision values. The resulting slowdown of those versions compared to the original single-precision implementations are shown in Table 3.1. The increase in execution time due to the use of double-precision values ranges from about 1.5x to 2.0x. These results

[5]The Texture Memory version could not be implemented using double-precision because the underlying hardware does not support double-precision textures.

suggest that the application is memory-bound rather than compute-bound, because the switch to double-precision doubles the application's memory bandwidth requirements.

For many applications, using single- instead of double-precision values can obviously impact the final results. For leukocyte detection, however, this turns out not to be the case. Most importantly, for the first frame of the video analyzed here, the number of cells detected by the single- and double-precision CUDA implementations (specifically the one-pass variance versions) are the same. More surprisingly, the x- and y-coordinates of those cells are equivalent out to 13 decimal places in both versions. This is more than enough precision for this application, since the final results are only meaningful in terms of integer pixel locations.

It might be objected that it is unfair to compare the performance of the single-precision CUDA implementations to the double-precision CPU implementations, and that perhaps we should use single-precision values in the CPU versions as well. However, it turns out that the performance of the CPU implementations do not improve if we use single-precision values in the GICOV computation and dilation steps. This is because, as in the CUDA implementations, the rest of the application uses double-precision values, and these inputs must first be cast to single-precision values before the computations can proceed. This overhead negates any speedup due to the faster single-precision arithmetic.

3.3 Accelerating the Tracking Stage

After the locations of leukocytes in frame i have been determined by the detection stage of the algorithm, this information is used by the tracking stage to determine the new locations of those same leukocytes in frame $i + 1$. These updated locations are then fed back into the tracking stage to determine the new cell locations in frame $i + 2$. This process continues, with detection typically performed once every 10 frames.

In each frame, all cells can be processed independently. For each cell, the algorithm only analyzes a fixed-sized portion of the frame (41x81 pixels for the particular leukocytes studied

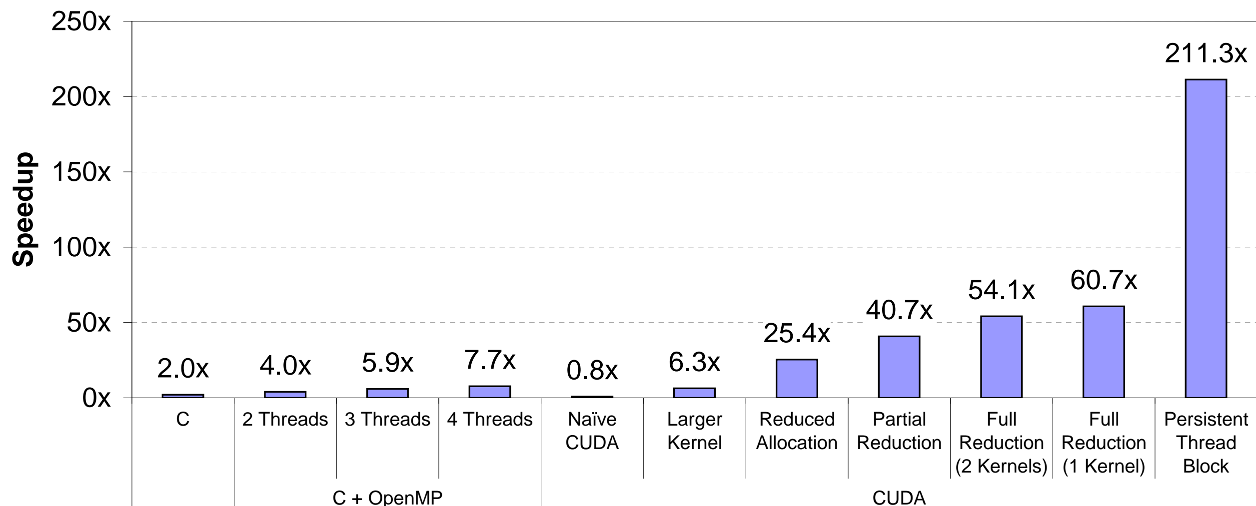


Figure 3.3: Speedup of the different implementations of the tracking stage over the original MATLAB implementation.

in this work), centered around the cell's location in the previous frame. This explicitly limits the maximum velocity at which a cell can be successfully tracked. Within the sub-image of interest, two operations are performed. First, the Motion Gradient Vector Flow (MGVF) matrix is computed via an iterative Jacobian solution procedure. The solver iterates until it has met a convergence criterion, which is a function of all of the elements in the matrix. Second, an active contour minimizes an energy function defined on the MGVF matrix and computes the new location of the leukocyte.

In the original MATLAB implementation of the tracking stage, 93.5% of the execution time is spent in the iterative solver. In the C implementation, the iterative solver consumes essentially all (99.8%) of the overall execution time. The C implementation provides a speedup of 2.0x over the MATLAB implementation. As with the detection stage, we further accelerated the tracking stage using OpenMP and CUDA. The execution time of each of the different implementations was measured on the same system as described in Section 3.2. The speedups achieved by the C, OpenMP, and CUDA implementations over the original MATLAB implementation are shown in Figure 3.3.

3.3.1 Comparison to OpenMP

Accelerating the tracking stage with OpenMP was a relatively straightforward transformation. Since each cell being tracked can be processed in parallel, we simply added a `parallel` pragma to the `for` loop that iterates over all of the cells. Because the number of cells is small (generally less than 50), this approach would not be effective if we attempted to scale the OpenMP implementation to a much larger number of processors. For the hardware on which we benchmarked the implementation, however, the decomposition was good enough to achieve nearly linear scaling. The OpenMP speedups for two, three, and four threads are shown in Figure 3.3. With four threads, the OpenMP implementation achieves speedups of 7.7x and 3.8x over the original MATLAB and C implementations, respectively.

3.3.2 Naïve CUDA Implementation

Because the execution time of the tracking stage is dominated by calls to the iterative solver, which in turn is dominated by calls to a regularized version of the Heaviside function, the first CUDA implementation simply replaced each call to the Heaviside function with a call to a Heaviside CUDA kernel. In this implementation, each element in the output matrix is computed by a single thread. Although the overall kernel execution time is slightly less than one second, the memory allocation and copying overheads add more than eleven seconds to the overall execution time. Due to these overheads, this implementation achieves a 2.6x *slowdown* compared to the original C implementation (and is actually slower than the MATLAB implementation). Parallelizing the OpenMP implementation at the granularity of individual calls to the Heaviside function similarly resulted in a significant slowdown.

3.3.3 CUDA Optimizations

As with the detection stage, a number of optimizations were applied to the naïve CUDA implementation of the tracking stage in order to improve its performance. For each implemen-

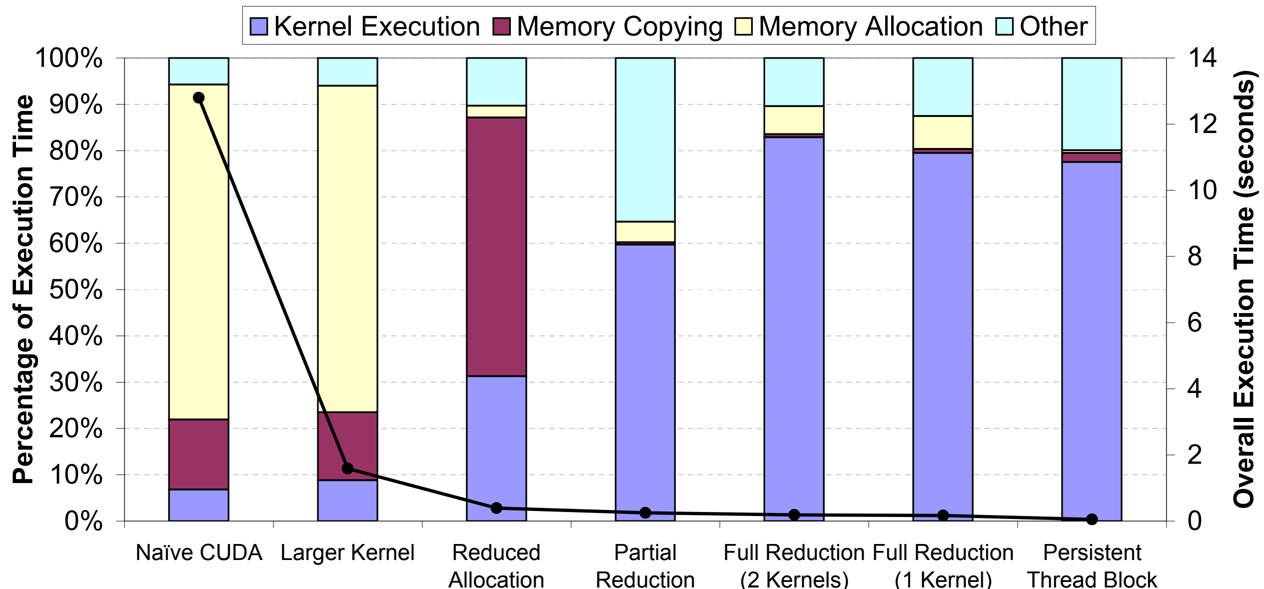


Figure 3.4: Impact of different overheads on the execution times of the CUDA implementations of the tracking stage. Each bar shows, starting at the bottom, the percentage of execution time due to: executing the CUDA kernels, transferring memory between the CPU and GPU, allocating memory on the GPU, and executing the other, non-CUDA related code. The line indicates the overall execution time of each implementation.

tation, Figure 3.4 shows the overall execution time, as well as the fraction of the execution time devoted to kernel execution, memory copying, memory allocation, and non-CUDA related code. Note that the optimizations are again cumulative, but unlike in the detection stage, they are mostly dependent on one other, since they change how and when memory is allocated and when data is moved to and from the GPU.

Larger Kernel: In the naïve implementation, the Heaviside kernel is called eight times during each iteration of the solver. In order to reduce the memory allocation and copying overhead and the number of kernel calls as well as increase the amount of useful work performed in each kernel call, the entire body of the inner loop was converted into a single CUDA kernel. As in the previous implementation, each element in the output matrix is computed by one thread. Applying this optimization yields an overall speedup of 3.1x over the original C implementation and 8.1x over the previous CUDA implementation. Recall that the OpenMP implementation is parallelized across the cells being tracked; if we instead

parallelize it across the individual matrix elements in the iterative solver, as is done here for the CUDA implementation, the OpenMP implementation actually becomes 24% slower.

Reduced Allocation: Allocating and deallocating memory on the CPU via the C standard library functions `malloc` and `free` is a relatively low overhead operation. Allocating and deallocating memory on the GPU via the CUDA library functions `cudaMalloc` and `cudaFree`, however, is considerably more expensive. On the system used in this study, we measured the overhead of `cudaMalloc` to be approximately 30-40 times greater than the overhead of `malloc` (and significantly higher for memory sizes larger than a few megabytes) and the overhead of `cudaFree` to be approximately 100 times greater than the overhead of `free`. This overhead is readily apparent in Figure 3.4 for both the naïve CUDA and larger kernel implementations, whose execution times are dominated by memory allocation.

In order to minimize this overhead, instead of allocating and freeing memory on the GPU once each iteration of the solver, initialization and cleanup functions were added to allocate memory a single time at the start of the iterative solver and then free memory at the end. Applying this optimization yields an overall speedup of 12.6x over the original C implementation and 4.0x over the previous CUDA implementation. Note that even if the C standard library memory allocation functions were as expensive as the CUDA equivalents, the overhead would be negligible in the OpenMP implementation because it does not allocate memory within the iterative solver loop.

Partial Reduction: After each iteration of the solver, the average of the absolute value of the change of each matrix element is computed in order to check for convergence. In the previous CUDA implementation, the entire MGVF matrix is copied back after each iteration, and the reduction is performed entirely on the CPU. In order to improve the performance of the reduction, the kernel was extended to perform a partial reduction, in which each thread block[6] computes the sum of the absolute value of the change of each matrix element within that thread block. With a thread block size of N threads, this reduces by a factor of N

[6] A *thread block* is the CUDA equivalent of an OpenCL work group.

both the amount of memory copied from the GPU to the CPU as well as the number of additions required by the CPU to perform the reduction. Since typical values of N in CUDA applications are 128 and 256, performing a partial reduction on the GPU can result in a substantial performance improvement. In this application, applying the optimization yields an overall speedup of 20.3x over the original C implementation and 1.60x over the previous CUDA implementation. This and the next two optimizations do not apply to the OpenMP implementation because it does not transfer data between disjoint memory spaces.

Full Reduction (2 Kernels): In order to further reduce the reduction and memory copying overheads, a second CUDA kernel was added to complete the reduction on the GPU. This allows the copying of the partial sums to be replaced by the copying of a single Boolean value indicating whether or not the computation has converged. However, this approach does not improve performance significantly because, although it does reduce the amount of data copied, it does not reduce the number of copies performed. At data sizes less than about four kilobytes, the latency of a memory transfer is essentially constant regardless of the amount of data transferred.

Thankfully, performing the entire reduction on the GPU enables a further optimization. Instead of having the host code on the CPU check the convergence flag after each iteration, the computation kernel can be modified to check the value of the flag and exit if convergence has already been achieved. This allows the computation and reduction kernels to be called as many times as desired without the need to explicitly copy the convergence flag and without impacting the correctness of the results. In other words, this allows the main loop of the iteration to be unrolled to an arbitrary degree. In our experiments, performing about 30 back-to-back kernel calls before copying the convergence flag resulted in the best performance. Applying these optimizations yields an overall speedup of 26.9x over the original C implementation and 1.33x over the previous CUDA implementation.

Full Reduction (1 Kernel): Although the previous optimization reduces the overall execution time, it actually increases the absolute execution time devoted to kernel execution,

due to both the increase in computation performed by the kernels as well as the doubling of the number of kernel calls. To reduce the kernel overhead, the computation and reduction kernels can be merged into a single kernel. However, we must be careful about the ordering of the computation and reduction in the merged kernel. A seemingly reasonable approach would be to compute the updated MGVF matrix at the beginning of the kernel and then perform the reduction at the end of the kernel. Unfortunately, this would require the use of a global memory fence in order to ensure that all thread blocks had finished their computations before the reduction was performed, and CUDA does not provide such a fence except across kernel calls.

To avoid this potential deadlock, in each kernel call we first perform a reduction on the values produced by the previous kernel call. Only then do we proceed to compute the next iteration (if the computation has not already converged). Applying this optimization yields an overall speedup of 30.2x over the original C implementation and 1.12x over the previous CUDA implementation.

Persistent Thread Block: In the previous implementation, about 24% of the time spent by the application waiting for kernel execution is due to the overhead of kernel invocation, with only 76% of the time due to actually performing useful work on the GPU. To reduce the overhead of kernel execution, we can perform all of the iterations in a single kernel call. As mentioned earlier, CUDA only provides a per-thread-block memory fence, not a global memory fence. Thus, in order to perform all of the iterations in a single kernel call, we must perform all of the computation for one cell within a single thread block. Since a single thread block can contain no more than 512 threads, and there are more than 3,000 elements in the MGVF matrix, we must abandon the one-to-one mapping between threads and matrix elements that is typically used in CUDA kernels. Instead, within each iteration, the single thread block traverses the entire matrix, computing a subset of the matrix in each step.

For performance reasons, it is desirable to maintain the current state of the matrix in the fast on-chip shared memory, rather than the slow off-chip global memory. Additionally, for

programming simplicity, it would be desirable to maintain two matrices, one containing the old state computed in the previous iteration and the other containing the new state being computed in the current iteration. These two copies would allow the thread block to produce the correct result regardless of the order in which it traverses the matrix. Unfortunately, shared memory on current NVIDIA GPUs is only 16 KB, and the matrix is more than 13 KB, so only one copy of the matrix can be maintained. To ensure that all computations only read matrix values from the previous iteration, the thread block traverses the matrix from the top down and writes the new values into a buffer (the size of the buffer is equal to the number of threads in a thread block). At the end of each step, each thread writes the value from the buffer into the matrix in the location from the previous step. Thus, the updates to the matrix are effectively delayed by one step, ensuring that every thread always uses the correct value from the previous iteration.

If we simply modify the kernel to perform all of the iterations for a cell in a single kernel call but still process the individual cells sequentially, the application will not effectively take advantage of the GPU's parallel computation resources and the resulting performance will be significantly worse than the previous implementation. However, since each cell now only requires a single thread block, it makes sense to process all of the cells concurrently, with one thread block allocated for each cell. The entire tracking stage for one frame can then be completed with a single kernel call. Implementing this optimization yields an overall speedup of 105.2x over the original C implementation and 3.5x over the previous CUDA implementation. Note that the OpenMP parallelization uses essentially the same approach, but with only a single thread processing each cell rather than an entire thread block.

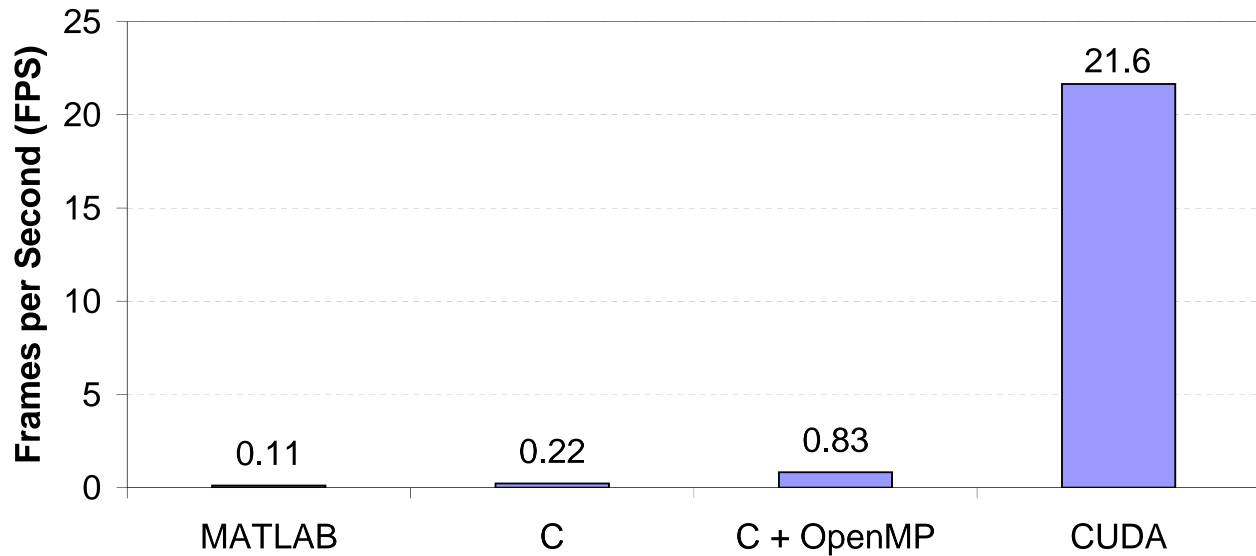


Figure 3.5: Overall rates at which the four implementations can detect and track leukocytes.

3.4 Discussion

The final CUDA implementation of the detection and tracking algorithm provides a speedup of 80.8x over the single-threaded C implementation[7]. Even assuming perfectly linear scaling, matching the performance of this CUDA implementation with the OpenMP implementation would require about 80 CPU cores equivalent to the cores used in our experiments. Given the choice to obtain the same speedup by purchasing either 20 quad-core processors (and associated hardware) or a single GPU, the most cost-effective choice is clearly the GPU. Of course, in practice we have been unable to achieve perfectly linear scaling with OpenMP on this problem due to the relatively small sizes of the computations involved, and in all likelihood we would be unable to match the performance of the GPU with any number of additional CPU cores.

All of the performance results presented so far have been expressed relative to the performance of other implementations. To provide a sense of how close each implementation comes to achieving real-time analysis, Figure 3.5 shows the number of frames of video

[7]To compute the performance of the entire application, we assume that detection is performed once every ten frames. Thus, the average time to process one frame can be estimated by $(D + 9T)/10$, where D and T are the average times to perform detection and tracking, respectively, on a single frame.

that each implementation can process per second. While the MATLAB, C, and OpenMP implementations cannot even process a single frame per second, the CUDA implementation can process more than twenty. Given the increases in GPU performance expected in the next few years, real-time detection and tracking of leukocytes at 30 FPS appears realizable in the near future with commodity hardware.

3.4.1 Lessons for Software Developers

Many of the difficulties we encountered in achieving good performance with the GPU were due to inherent assumptions that we made about the costs of certain operations. These assumptions break in the context of GPU computing: launching a kernel is approximately three orders of magnitude more expensive than calling a CPU-side function; GPU memory allocation functions are around three orders of magnitude slower than their CPU counterparts; transferring a single byte between the CPU and GPU is as expensive as transferring 8 kilobytes; global synchronization requires the costly invocation of an entirely new kernel; and slightly irregular access patterns that would be efficiently captured by a CPUs cache can lead to disastrous performance on the GPU. Achieving good performance in the face of these invalidated assumptions can require programmers to make non-trivial changes to the manner in which an application is parallelized. Later we will suggest ways in which system architects can significantly reduce or even eliminate some of these bottlenecks, but here we focus exclusively on techniques that allow CUDA application developers to bypass these bottlenecks to some extent.

Reduce Kernel Overhead: We have shown earlier that the overhead of launching a kernel can severely impact the performance of a CUDA application. This is clearly evident when we compare the performance of the naïve CUDA implementations of the two different stages of the algorithm. In the detection stage, the most natural decomposition was at a coarse-grained level, resulting in only two kernel calls per frame. In the tracking stage, however, the most natural decomposition was at a much finer-grained level, resulting in

approximately 50,000 kernel calls per frame. As a result, only 0.1% of the time spent waiting for the execution of the GICOV kernel in the detection stage is caused by the kernel invocation overhead, with 99.9% of the time spent performing actual computation on the GPU. Conversely, 73.1% of the time spent waiting for the execution of the Heaviside kernel in the tracking stage is caused by the kernel overhead, with only 26.9% of the time spent performing actual computation. Thus, regardless of how much we were able to improve the performance of the Heaviside kernel, we would not be able to reduce the overall execution time of the kernel by more than 26.9%. In order to reduce the impact of this overhead, developers should attempt to make their kernels as coarse-grained as is feasible, thereby increasing the amount of work performed in each kernel call and reducing the total number of kernel calls.

There is also a performance advantage due to launching many kernels back-to-back. For example, in both full reduction implementations of the tracking stage, the overhead of kernel invocation is significantly more severe without unrolling the iterative solver loop. This is because, in the most recent versions of the CUDA API, kernel invocations are asynchronous. With unrolling, multiple kernel calls are batched in the GPU driver, and the application can overlap kernel execution on the GPU with accessing the driver on the CPU. Without unrolling, there is an implicit synchronization when the convergence flag is copied back to the CPU after each kernel call, and there is no overlap between kernel execution and driver access.

Reduce Memory Management Overhead: As mentioned earlier, `cudaMalloc` and `cudaFree` are approximately 30-40 and 100 times more expensive, respectively, than `malloc` and `free`, their C standard library equivalents. The results for the naïve CUDA and larger kernel implementations of the tracking stage demonstrate this clearly. Allocating memory on the GPU consumes approximately 72% and 71%, respectively, of the execution times of those two implementations. The solution here is straightforward: wherever possible, allocate GPU

memory once at the beginning of an application and then reuse that memory in each kernel invocation.

Reduce Memory Transfer Overhead: Another inefficiency caused by the disjoint address spaces of the CPU and GPU is the need to explicitly transfer data between the two memories. The transfer overhead can be significant: in the reduced allocation implementation of the tracking stage, memory copying consumes 56% of the overall execution time. To reduce the severity of this overhead, developers should attempt to perform as much computation on the GPU as possible. For example, in the partial reduction implementation, the convergence condition is partially computed on the GPU in order to reduce the memory transfer overhead. With this change the number of elements transferred decreases from the number of elements in the matrix (generally 3,321) to the number of thread blocks (52 in this case). It is important for developers to understand that accelerating a computation using CUDA does not have to be an all-or-nothing proposition. Even if an entire computation cannot be (easily) implemented using CUDA, it is possible that offloading only a part of the computation (e.g., part of the reduction) may still improve the overall performance.

Note also that moving a computation to the GPU may prove beneficial even if that computation would be more efficiently executed on the CPU. To further reduce the memory copying overhead of the partial reduction implementation, the two-kernel full reduction implementation uses a second kernel to finish summing the partially reduced values produced by the first kernel. Even though the second kernel is invoked with only a very small number of threads, which certainly perform the reduction significantly slower than would a CPU thread, the change improves the application's overall performance because the reduction in the memory transfer overhead outweighs the increase in computation time. CUDA implicitly encourages developers to fill the GPU with thousands of threads, so that they are trained to think that they are wasting the GPU's computational resources if they use only a small number of threads. However, as we have seen here, it is sometimes advantageous to accept computational inefficiency in exchange for a reduction in memory transfer overhead.

Understand Memory Access Patterns: CPUs are designed to reduce the effective memory access latency through extensive caching that leverages the temporal and spatial locality present in most workloads. Thus, a slightly irregular memory access pattern, such as the one exhibited by the stencil operation used in the GICOV computation, can be successfully captured by the CPU's caches. However, that same access pattern may be irregular enough to prevent efficient utilization of the GPU's memory bandwidth, because the restrictions on access patterns that must be met in order to achieve good memory performance are much more strict on a GPU than they are on a CPU. This is evident in the GICOV kernel of the detection stage. In the original implementation, the input matrices are allocated in row-major order, so access would be most efficient if neighboring threads access neighboring elements from the same row. However, the access pattern actually exhibited by the kernel is that neighboring threads access neighboring elements from the same *column*. This explains why allocating the input matrices in column-major order provides a 2.8x speedup. The same change in the CPU version does not significantly impact the execution time because the caches are large enough to capture the entire stencil regardless of the order of traversal.

These access pattern restrictions can be partially relaxed by taking advantage of the GPU's special-purpose address spaces. Both constant and texture memory provide small on-chip caches that allow threads to take advantage of fine-grained spatial and temporal locality. In addition, texture memory relaxes the alignment requirements that must be met in order for multiple memory accesses from within the same warp to be coalesced into a single memory transaction. Another effective approach is to use the software-controlled shared memory as an explicitly managed cache, which can significantly improve performance when data elements are frequently reused among threads in the same thread block.

Tradeoff Computation and Memory Access: The GICOV and dilation kernels used in the detection stage perform relatively simple computations across a large number of data elements. Thus, their performance is more a function of the GPU's memory system performance than its processing performance. It can be beneficial for such memory-bound

kernels to decrease the number of memory accesses required by increasing the complexity of the computation. Such a case arises in the GICOV kernel, which at each pixel and for each stencil computes the variance of a function across the 150 sample points within that stencil. The original CUDA implementation computes the variance in two passes. Since computing each point in the function requires accessing global memory, implementing a single-pass algorithm for computing the variance essentially halves the number of memory accesses. Even though the single-pass algorithm significantly increases the complexity of the variance computation, it provides a 1.4x speedup over the two-pass algorithm because the impact of the reduction in memory usage far outweighs the impact of the increased computational complexity. Similar transformations are likely to be possible for other memory-bound kernels.

Avoid Global Memory Fences: As discussed earlier, CUDA does not provide a global, inter-thread-block memory fence. Thus, if multiple thread blocks need to communicate, they must do so across kernel calls. This would not present a problem if the overhead of kernel invocation were not so high. In the two-kernel full reduction implementation of the tracking stage, a global memory fence is needed in each iteration between the matrix computation and the convergence check. This fence is implemented by creating separate kernels for the two steps. Unfortunately, this doubles the number of kernel calls, which limits the overall performance. As described earlier, one technique for reducing the number of kernel calls is to switch the order of the two steps and combine them into a single kernel, so that the convergence check occurs before the matrix computation in each iteration. Although this introduces redundant computation, since the final step in the reduction is performed by each thread block instead of by a single thread block, the reduction in the kernel overhead produces an overall speedup of 1.12x over the two-kernel implementation. This technique is generally applicable to any iterative solver that uses a convergence criterion for early exit.

Although this approach reduces the number of kernel calls by a factor of two, it still requires the use of a global memory fence after each iteration. This is because there is a one-to-one mapping between threads and matrix elements, and the number of matrix

elements is larger than the maximum size of a thread block. The thread mapping scheme used here is typical in CUDA programs, because CUDA developers are encouraged to make their threads as fine-grained as possible in order to fully utilize the GPU's vast computational resources. However, abandoning this canonical thread mapping and instead using only a single thread block allows an arbitrary number of iterations to be computed in a single kernel call without the need for a global fence. As long as there are enough independent computations (corresponding to individual cells in this work) to occupy most or all of the cores, this approach can provide significant speedups. Note also that the performance advantage increases as the number of iterations of the solver increases. Thus, the slower the computation converges, the more advantageous it becomes to use a single, persistent thread block for each independent computational unit.

3.4.2 Lessons for System Architects

We have demonstrated techniques for avoiding or mitigating many of the performance bottlenecks that a CUDA developer may encounter. A more effective approach, however, would be for system designers to reduce the impact of those bottlenecks or avoid introducing such bottlenecks altogether. We suggest a number of approaches that a system architect can take, at both the hardware and software levels, to reduce the amount of effort required for developers to obtain satisfactory performance. Removing some of the barriers to high performance will help speed the adoption of CUDA and other GPGPU programming models.

Streamline Memory Management: Perhaps the simplest bottleneck to address would be the slow memory management provided by the CUDA API. As noted earlier, the `cudaMalloc` and `cudaFree` functions are significantly slower than the equivalent C standard library functions, `malloc` and `free`. If the CUDA memory allocation functions were as fast as the equivalent C standard library functions, the larger kernel implementation of the tracking stage would provide a 2.5x speedup over the best OpenMP implementation instead of the 1.2x *slowdown* that it actually provides. Thus, with a relatively straightforward

translation to CUDA and without any complex optimizations, this CUDA implementation would have been adequate to provide better performance than the best CPU implementation. Reducing the overhead of memory management would both simplify the process of achieving satisfactory speedups with simple implementations and enable even more impressive speedups with complex implementations.

The inefficiency of memory allocation may be a byproduct of the fact that most graphics applications tend to allocate memory both in large chunks and on an infrequent basis. Thus, there traditionally has been little incentive for the authors of graphics drivers to optimize the memory management functions. With increased adoption of CUDA and other GPGPU programming models, it becomes more important to address these inefficiencies.

Provide a Global Memory Fence: CUDA's lack of an inter-thread-block global memory fence forced us to use a non-intuitive implementation strategy in order to achieve the most significant speedup on the tracking stage. The use of a persistent thread block runs counter to the standard CUDA development strategy of making threads as fine-grained as possible. If CUDA provided an inter-thread-block memory fence, the full reduction implementation could have achieved significantly better performance without the need to abandon the one-to-one mapping between threads and matrix elements. Assuming that the overhead of the fence would be negligible in comparison to the overhead of the computation itself, using a memory fence in the full reduction implementation instead of multiple kernel calls would speed up that implementation by 1.3x.

Without detailed knowledge of the GPU's microarchitecture, it is difficult to assess the complexity of implementing a global memory fence. One required change is clear, however. In the general case, implementing a global fence in CUDA would require thread blocks that reach the fence to yield to thread blocks that are still waiting to begin execution, in order to ensure forward progress when there are more thread blocks than can execute concurrently on the GPU. Yielding a thread block would require each thread to write its current state to memory. For small numbers of thread blocks, this would be relatively inexpensive. However,

the CUDA specification allows a kernel to be invoked across more than four billion thread blocks of up to 512 threads each. Clearly the GPU's memory would not be large enough to store the state for so many threads, and thus an application using a global memory fence would require a much lower limit on the number of threads per kernel invocation. For many applications, this would be an acceptable tradeoff.

Add Caches: The GPU's use of on-chip caches for the constant and texture memory spaces allows developers to achieve good memory performance for some read-only data structures even with kernels whose memory access patterns are slightly irregular. Unfortunately, in order to achieve good memory performance with data structures allocated in the global memory space, the access pattern restrictions are much more severe. Thus, for data structures that need to be updated and which are unsuitable for the on-chip shared memory, there is a significant burden placed upon developers to meet those restrictions. The introduction of a relatively modest amount of on-chip cache for the read-write global memory space would substantially reduce the burden on developers of ensuring the regularity of a kernel's memory accesses, at the expense of raising coherence issues.

Add a Control Processor: A more substantial architectural change would be to add to the GPU a small control processor that provides higher single-thread performance than the underlying throughput-oriented cores. If this core were able to launch kernels, then the overhead of kernel invocation would be significantly decreased since the latency between the control processor and the parallel substrate would be much lower than the latency between the CPU and the substrate. Additionally, applications with non-trivial sequential phases could be efficiently supported in a more straightforward manner. For example, the reduced allocation implementation of the tracking stage performs one iteration of the solver on the GPU and then transfers the current state of the matrix back to the CPU to perform the reduction and check for convergence. Copying the matrix from the GPU to the CPU consumes more time than the actual kernel execution. If the serial reduction could instead be executed on the GPU's control processor, this memory transfer overhead could be avoided.

3.5 Related Work

The availability of cheap, high-performance GPUs which can be programmed using a familiar programming abstraction has led a large number of developers to port their applications to CUDA. Garland et al. [37] provide a good overview of the experiences and speedups achieved in a number of application domains. Many developers are working with applications that are more naturally ported to CUDA because they consist of kernels that perform huge amounts of work. These developers do not encounter many of the overheads associated with fine-grained kernels that we explore in this work. Only a few have fully explored the optimizations necessary to obtain significant speedups.

Automating the exploration of CUDA configurations in order to optimize performance was explored by Ryoo et al. [90]. The authors of that work do not consider mapping major data structures to different memory spaces in CUDA or reorganizing their memory layout to achieve higher performance, and do not explore more complex optimizations such as trading off the amount of computation done on the CPU and on the GPU. The optimization strategies of multiple applications and the use of CUDA's rich memory hierarchy were explored by Che et al. [23]. However, they focus on applications which have a large amount of work per kernel call, and thus do not have to deal with the system bottlenecks explored in this work.

3.6 Conclusions

We have shown that leukocyte detection and tracking can benefit greatly from using a GPU. The algorithms used in the detection and tracking stages, namely stencil computations and iterative solvers, are also used in a wide range of other application domains, which can all benefit from the optimizations we have discussed. Overall, the best CUDA implementation provides speedups of 58.5x and 211.3x on the detection and tracking stages, respectively, over the original MATLAB implementation and 9.4x and 27.5x over the best OpenMP implementation. While the MATLAB implementation takes more than four and a half hours

to process one minute of video, the CUDA implementation can process that same video in less than one and a half *minutes*. Put another way, while the MATLAB implementation can detect and track leukocytes at 0.11 FPS, the CUDA implementation operates at 21.6 FPS. For video recorded at 30 FPS, continued scaling of hardware resources means that real-time analysis is almost within reach for inexpensive workstations.

While straightforward CUDA implementations can achieve substantial benefits, especially with a modest amount of tuning, significant programmer effort can be required to make full use of the GPU's potential when irregular memory access patterns or small kernels are present. Despite this extra effort required to realize the potential of the GPU, the benefits can be dramatic. Our experiences with CUDA show the power of the GPU as a parallel platform, and help demonstrate how the variety of many-core platforms that we expect to see in the future will transform computational science.

3.7 Postscript

As noted earlier, the work presented in this chapter was completed in early 2009. Since that time, NVIDIA has introduced significant hardware and software changes. In 2010, NVIDIA introduced a new GPU architecture, Fermi, and followed up with another new architecture, Kepler, in 2012. On the software front, NVIDIA has released three new major versions of CUDA.

To better understand the impact of the hardware and software improvements that have taken place over the last four years, we measured the performance of the leukocyte tracking application on one of the highest-performance CPUs and one of the highest-performance GPUs available as of early 2013. We ran the application on a 16-core, 2.6-GHz AMD Opteron 6282 SE CPU, which has four times as many cores as the CPU used in the results presented earlier. The overall CPU performance improved by 2.8x from 0.83 FPS to 2.33 FPS, which is still 12.9x slower than the 30 FPS required for real-time processing. The improvement

in CPU performance is due to the increased parallelism (i.e., the larger number of cores); the performance of the serial C implementation is essentially unchanged. We also ran the application on a NVIDIA Tesla K20, a Kepler-series GPU. The overall GPU performance improved by 1.9x from 21.6 FPS to 41.4 FPS, easily achieving real-time processing. Overall, the GPU's performance advantage relative to the CPU has narrowed from 26.0x to 17.8x.

Many of the system-level changes we suggested in Section 3.4.2 have since been implemented in one form or another. The changes, and their impact on the optimization strategies discussed earlier, are as follows:

- A global memory fence instruction[8] was introduced in CUDA 2.2.1 in May 2009 [73]. We can leverage this instruction to complete the tracking stage in a single kernel call (per cell) without resorting to the relatively complex persistent thread block implementation. Preliminary investigations suggest that such an implementation can provide significantly better performance than the fastest non-persistent-thread-block implementation, but is still significantly slower than the persistent thread block implementation.

- Readable and writeable L1 and L2 caches were introduced in the Fermi architecture and have persisted in Kepler. The addition of these general-purpose caches has made the use of the special-purpose address spaces (constant and texture memory) less necessary for high performance for many kernels.

- Although NVIDIA has not added a control processor to their GPUs, they have added the ability for one kernel to launch another kernel. This capability could potentially be leveraged to reduce the overhead of launching many kernels back to back. However, launching multiple kernels in this way loses one of the advantages of using separate

[8]Note that this new instruction is not a global *barrier*, as we assumed when discussing a hypothetical global memory fence instruction in Section 3.4.2. Instead, once this instruction returns, all of the writes to global memory by the calling thread are guaranteed to be visible to all other threads. This instruction could be used in conjuction with other instructions to implement a global barrier, although such an implementation would be subject to the same limitations we discussed earlier.

kernel invocations: the implicit global memory fence between successive kernels. This limitation can be overcome, but it significantly increases the implementation effort.

In addition, a number of architectural changes unrelated to the suggestions we made earlier have also been introduced. For example, both the Fermi and Kepler architectures include larger shared memories than previous architectures. The persistent thread block implementation of the tracking stage could take advantage of this increased capacity to process larger cells and/or use a more natural double-buffering implementation with two full copies of its matrix.

A major new version of the PCIe standard, version 3.0, was released in November 2010, although the first GPU supporting PCIe 3.0 (the AMD Radeon HD 7970) was not released until January 2012. Relative to PCIe 2.0, PCIe 3.0 approximately doubles the peak transfer throughput between the CPU and the GPU. The latency for small transfers, however, is essentially unchanged. Because the leukocyte tracking application mostly uses small transfers, the benefits of upgrading to a system (motherboard and GPU) supporting PCIe 3.0 would be minimal. Additionally, the overhead of allocating memory and launching kernels has changed little in the past few years. Almost all of the advice we provide for developers in Section 3.4.1 is still quite relevant. Much of this advice can now be found in NVIDIA's Best Practices Guide [74], which was first published in July 2009.

Most of the overhead of launching a kernel on a GPU is due to software overheads. A kernel dispatch request must be processed by multiple levels of software: the CUDA or OpenCL runtime, the user-mode graphics driver, and finally the kernel-mode graphics driver. Accessing the kernel-mode graphics driver is particularly expensive. The proposed Heterogeneous System Architecture (HSA), being developed by AMD and other member companies of the HSA Foundation, aims to reduce this overhead by allowing user-mode kernel dispatch for integrated GPUs [52]. This change will not help applications running on discrete GPUs, like the GPU used in this work, but it may make integrated GPUs significantly more attractive for iterative applications.

3.8 Publications and Impact

The work presented in this chapter was published at the International Parallel and Distributed Processing Symposium (IPDPS) in May 2009 [19]. Both CPU and GPU versions of the leukocyte tracking application have been released as a plugin [13] for the ImageJ image processing system. Serial, OpenMP, CUDA, and OpenCL versions of the application are included in the Rodinia benchmark suite [54]. Further analysis of the Rodinia applications, including the leukocyte tracking application, was published at the International Symposium on Workload Characterization (IISWC) in both 2009 and 2010 [22, 24]. The Rodinia suite has been downloaded more than 1,000 times and cited by other researchers more than 200 times. The leukocyte tracking application, along with other applications in the Rodinia suite, has also been identified by the SPEC High-Performance Group (HPG) for potential inclusion in a proposed accelerator benchmark suite.

Chapter 4

Heterogeneous Load Balancing

Most applications running on heterogeneous, multi-device systems, including the leukocyte tracking application discussed in the previous chapter, target only the single most powerful device, leaving other devices idle and potentially wasting much of the available computational power. Unfortunately, developing an application that can utilize all available devices effectively, and do so consistently across a wide range of diverse systems, is extremely challenging. In this chapter, we formulate this challenge as a rate matching problem: we must match the rate at which we send work to each device to the rate at which each device can complete that work. What makes this problem especially difficult is that a device's execution rate can vary widely, both across different applications and within the same application as the state of the system changes.

Prior work has developed load-balancing frameworks that automatically divide work across the available devices with little or no extra programmer effort [50,61]. However, these existing frameworks either assume that all devices in the system provide equal performance [50] or require a series of offline training runs to determine their relative performance [61]. In this chapter, we propose a dynamic scheduling algorithm that supports heterogeneous hardware and requires no offline training. In addition, our proposed scheduler is able to respond to dynamic performance fluctuations that occur at run time, such as those caused by changes

to a device's clock frequency. The algorithm first estimates each device's execution rate by scheduling a subset of the available work on each device. It then schedules the remaining work based on this initial estimate. However, by the time it has enough information to schedule the remaining work, there will likely be some work already scheduled but not yet completed, so the algorithm must also predict the expected completion time of this work and take that into account when scheduling.

We focus our attention on applications that execute a single data-parallel kernel at a time, with the output of the kernel consumed by the host program. Previous work has described effective approaches for load-balancing applications with multiple concurrently executing kernels [9, 12, 32, 38, 98, 103] or applications in which a single kernel is run repeatedly and consumes its own output [1, 26]. The challenge in the former case is determining on which device to execute the entirety of each kernel, while the challenge in the latter case is refining the work partition in each iteration while taking into account data locality. Solutions to these challenges do not help in effectively load-balancing the single-kernel applications that we consider in this chapter.

Although our approach to load balancing is general enough to apply to applications implemented in many programming languages, here we explore its utility in the context of OpenCL. We show that our dynamic approach provides consistently good performance: compared to the best possible static partition, it is on average 9.6% faster in dynamic conditions and only 2.2% slower in static conditions, without the costly training required by a real static approach. We also show that our algorithm outperforms existing dynamic scheduling algorithms designed for load balancing in multi-processor machines.

4.1 Motivation

To motivate the need for heterogeneous load balancing in general and dynamic load balancing in particular, we use as a case study two OpenCL applications from the AMD Accelerated

Configuration	Time (s)	Efficiency
Integrated GPU	0.95	44%
Discrete GPU	0.74	56%
Load Balancing	0.43	97%

Table 4.1: Average execution time and efficiency of Matrix Multiplication for native single-device execution and load balancing. Load balancing uses a fixed partition with 56% of the work assigned to the discrete GPU and 44% to the integrated GPU.

Parallel Processing (APP) SDK [4]: Matrix Multiplication and Discrete Cosine Transform (DCT).

4.1.1 Need for Heterogeneous Load Balancing

In the traditional GPGPU model, sequential or task-parallel control code is run on the CPU and performance-critical data-parallel code is run on a single GPU. In a system with a powerful multi-core CPU or multiple GPUs, this model fails to utilize the available resources fully and wastes much of the system's overall performance potential. Such systems are becoming increasingly common as microprocessor vendors incorporate accelerators onto CPU dies (e.g., AMD Accelerated Processing Units or APUs and Intel Sandy Bridge) and portable systems are configured with separate integrated and discrete GPUs to provide a tradeoff between energy efficiency and performance (e.g., AMD Radeon Dual Graphics and NVIDIA Optimus).

Table 4.1 shows the performance of Matrix Multiplication running on a machine with two GPUs, one discrete and one integrated. For single-device execution, the discrete GPU was 23% faster than the integrated GPU but reached only 56% efficiency[1]. By dividing the work across both devices, we further improved performance by 42% and reached 97% efficiency. The potential impact of load balancing depends on the relative performance of the devices in the system: the smaller the performance gap, the larger the potential benefit. However, even

[1]We define *efficiency* as the measured throughput divided by the sum of the throughputs of the devices when executing separately.

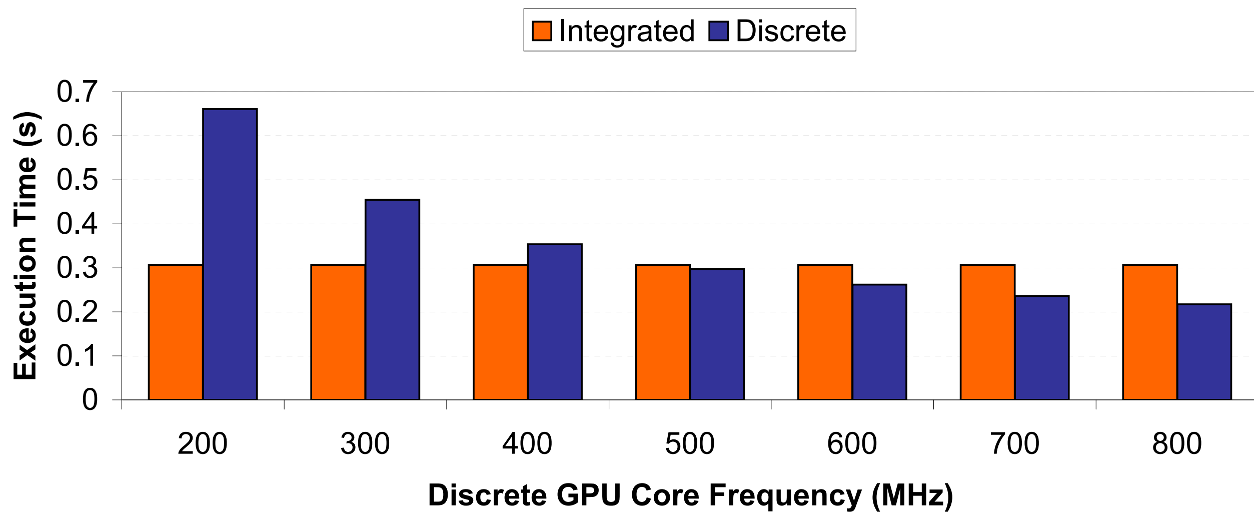


Figure 4.1: Average native, single-device execution time of DCT as the core clock frequency of the discrete GPU is adjusted.

with relatively large differences in performance, load balancing can still provide non-trivial speedups.

4.1.2 Need for Dynamic Load Balancing

The optimal division of work depends on the relative computation rates of the devices in a system, which can vary significantly at run time due to system- or application-level changes. For example, power or thermal constraints may force the system to scale down the clock frequencies of one or more devices, or contention from another application may decrease the performance of one of the devices. Both throttling and contention may occur more frequently in systems with multiple computational devices integrated into a single package and sharing a single memory system and power budget, as is the case in an AMD APU.

To demonstrate the impact of dynamic performance variation, Figure 4.1 shows the native execution time of DCT as we scaled the core clock frequency of the discrete GPU from 800 MHz to 200 MHz. Over this range, the discrete GPU's execution time increased by 3.04x. More importantly, the discrete GPU's performance relative to the integrated GPU fell from a 1.41x speedup to a 2.16x *slowdown*.

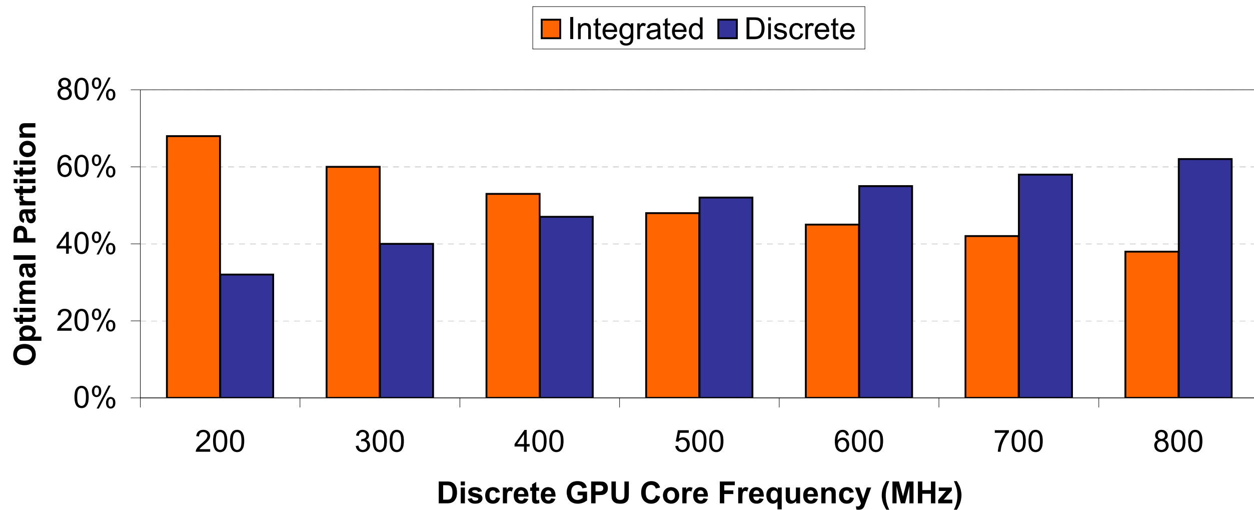


Figure 4.2: Optimal partition of work between the discrete and integrated GPUs for DCT over a range of discrete GPU clock frequencies.

Due to this large variability in relative performance, any fixed partition of work that provides good performance at one frequency will necessarily perform poorly at a different frequency. Figure 4.2 shows the optimal partition of work at each frequency, which varies from 62% discrete at the maximum frequency to 32% discrete at the minimum frequency. Note that these partitions were discovered via exhaustive search; in practice, partitions discovered via more reasonable means may be less efficient.

Figure 4.3 shows the execution time of three of these partitions across the entire frequency range, normalized to the execution time of the best partition at each frequency. No partition did consistently well; the best partition on average was the one optimized for 400 MHz, but it was 13% slower than the optimal in aggregate and 39% slower in the worst case. If we do not know a priori what frequency will occur most often, or if the set of expected frequencies spans a wide range, the partition we choose may be far from optimal.

In principle, we could attempt to construct a model that predicts the optimal partition based on the current clock frequency. Unfortunately, this further increases the training overhead (and may assume capabilities, such as the ability to adjust clock frequencies, that are inaccessible to the training infrastructure). Also, frequency scaling is just one source of performance variability, and it may be difficult or impossible to anticipate all other sources.

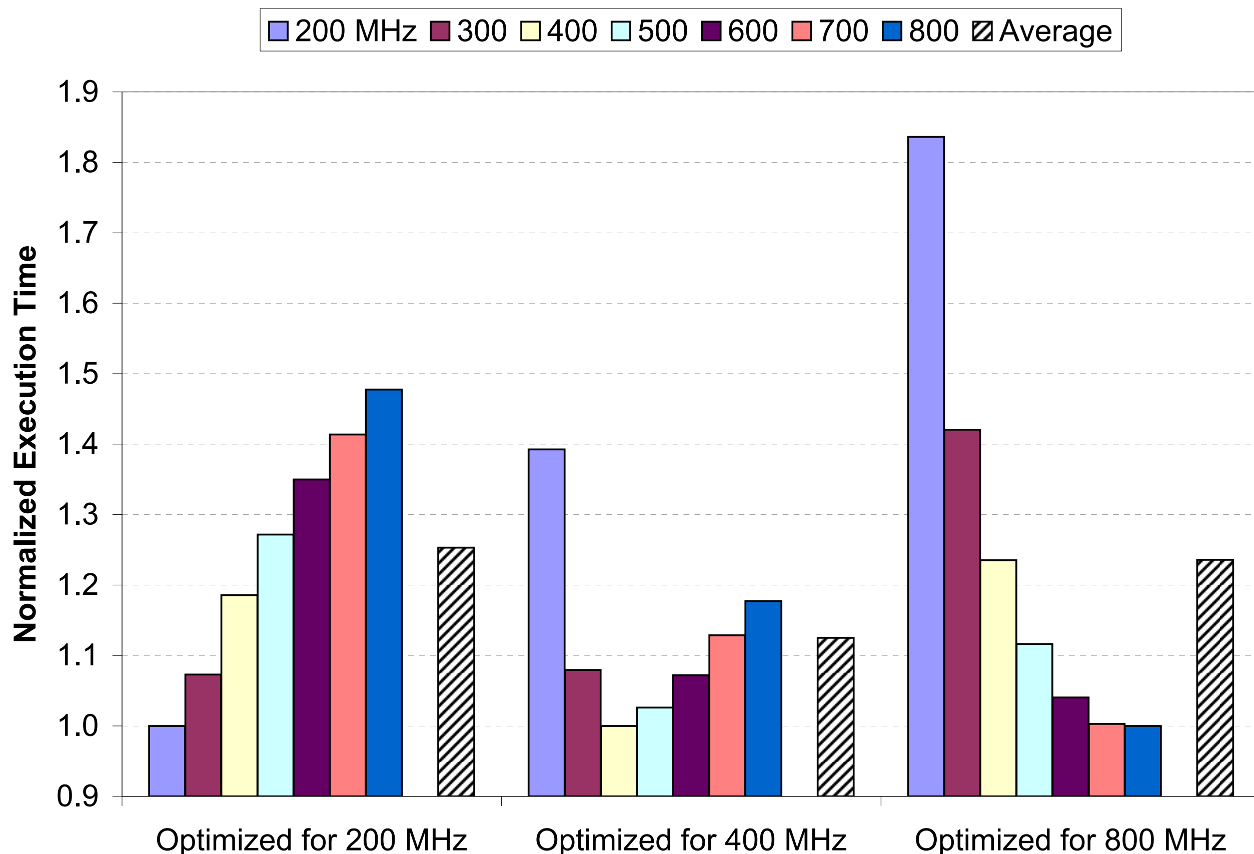


Figure 4.3: Average execution time of DCT for three fixed partitions optimized for different frequencies, normalized to the best partition at each frequency.

Even sources that we do anticipate (such as contention from other applications) may be significantly more difficult than frequency scaling to measure and account for.

Instead, we propose a dynamic load-balancing approach in Section 4.3 that can respond to performance variability regardless of its underlying cause. We show in Section 4.5.2 that such a dynamic approach can effectively respond to frequency scaling without any special knowledge or awareness of the underlying cause of the performance variability.

4.2 Related Work

Previous approaches to load-balancing a single kernel differ significantly in the amount of training they require. The simplest approaches, such as those proposed by Kim et al. [50] and Moerschell and Owens [66], require no training because they target systems with homogeneous

GPUs and therefore use a fixed, homogeneous work partition. At the other extreme, Wang and Ren [105] proposed trying a large number of different work distributions across a CPU and a GPU to find the most efficient from either a performance or energy perspective. Other approaches, like Qilin [61] and systems proposed by Shei et al. [95] and Nere et al. [70], use more modest amounts of training to select the work partition.

All of these approaches generate static work partitions and are unable to respond to dynamic performance variability. Chen et al. [26] proposed a dynamic approach that uses a centralized task queue to load-balance across multiple GPUs. However, they focused only on load-balancing computation, not data; they copied all of the input data to each device and did not account for this overhead in their results.

Numerous dynamic load balancing approaches for cluster computing exist, such as Dynamic Resource Utilization Model (DRUM) [100] and Parallel Framework for Unstructured Meshes (ParFUM) [110]. However, adapting them for use in modern heterogeneous systems would be non-trivial because they assume that work can be migrated from one processing unit to another at any time, even after it has begun execution. This assumption fails in the context of GPU computing, where we must explicitly decide how much work a device will complete before we can make our next scheduling decision.

A number of recently proposed load-balancing systems support applications with multiple concurrent kernels [9,12,32,38,59,98,103]. These systems attempt to maximize performance by determining the best kernel-to-device mapping, either automatically or through programmer directives, but do not support dividing a single kernel across multiple devices. Our proposed system, on the other hand, supports either single- or multi-kernel applications. Acosta et al. and Ma et al. [1,62] have proposed load-balancing systems for iterative applications, in which the work partition is refined gradually over many iterations of an application. Both systems always begins with a homogeneous partition and thus do not efficiently support applications with only a single kernel invocation.

Many existing load-balancing approaches rely heavily on manual intervention by the programmer. Extensions to Cg and GLSL (Zippy [35]), CUDA (CUDASA [67]), Intel Threading Building Blocks (Merge [58]), and OpenCL (Sun et al. [98]) simplify the process of dividing either a single or multiple kernels across multiple devices, but all require the programmer to manually specify the work partition. Same Program for All Processors [44] partitions work semi-automatically, but still requires the programmer to specify manually the expected relative performance of the devices in the system for a given application. Our proposed load-balancing algorithm does not require the programmer to aid in partitioning.

Another possible approach to heterogeneous scheduling would be to leverage self-scheduling algorithms, which were originally proposed for load balancing in large-scale multi-processor machines. In a self-scheduling algorithm, each processor or device dynamically claims a chunks of work; once it completes one chunk, it claims another chunk, repeating this process until no work remains. The self-scheduling algorithms differ in how the size of the chunks of work change over time: chunk self-scheduling [51] uses a fixed chunk size, guided self-scheduling [80] uses an exponentially decreasing chunk size, and trapezoid self-scheduling [104] uses a linearly decreasing chunk size. All three assume that the underlying hardware is homogeneous and that the primary source of performance heterogeneity is the workload itself. In this chapter, we consider applications and systems in which the heterogeneity of the hardware vastly outweighs any heterogeneity in the workload, and our proposed scheduling algorithm is designed explicitly to account for this. In Section 4.5, we compare the performance of these self-scheduling algorithms to the performance of our proposed algorithm.

AMD's CrossFire [5] and NVIDIA's SLI [72] support the automatic load balancing of graphics applications across two or more GPUs. With these approaches, each GPU either renders alternate frames or a portion of each frame (e.g., the top half or the bottom half). These approaches take advantage of special properties of graphics applications and typically rely on special hardware connections between the GPUs to achieve maximum performance. They also are limited in the amount of heterogeneity they can support: they typically do not

support GPUs from different generations nor GPUs from different market segments within the same generation (e.g., a high-end and a low-end GPU). These approaches do not support general-purpose kernels, and the algorithms they employ for load balancing are likely too special-purpose to be worthwhile adapting for general-purpose use.

4.3 Dynamic Load Balancing

Given an OpenCL application and kernel targeting a single compute device, the goal of our proposed load-balancing approach is to partition the kernel efficiently into *chunks* of contiguous work groups and schedule those chunks for execution across multiple devices. Previous work described the necessary mechanisms for intercepting and transforming OpenCL API calls to support multi-device execution, automatically determining which data is required by a given subset of the kernel, and determining in which dimension to partition a multi-dimensional domain [50]. Here we are concerned only with the actual scheduling of the kernel executions and data transfers that comprise a chunk.

Informally, our scheduling algorithm works by sending a small portion of the available work to each device and then using the execution time of that initial work to partition the remaining work. Before presenting the details of the algorithm in Section 4.3.2, we first derive equations for estimating the optimal partition of the remaining work based on the performance of the initial chunks. We focus on load balancing across two devices; however, the analysis presented in this section is easily extended to more than two devices.

4.3.1 Optimal Partition

To estimate the optimal partition, we must be able to predict the time that will be required to complete the remaining, unscheduled work as well as the time required to complete any work that has already been scheduled but has not yet completed. For both quantities, the

critical parameter is the execution time per work group[2] for device i, denoted Ω_i. We assume that the performance observed for the most recently completed chunk is a good predictor of the performance expected for the next chunk because the amount of work per work group is constant in most GPU kernels; Section 4.5.3 explores the accuracy of this assumption. Thus, we compute Ω_i as the total execution time (including data transfer time) of the most recently completed chunk divided by the number of work groups in that chunk. We assume that the amount of data transferred and data transfer time is the same across all work groups; the algorithm could be trivially extended to support applications in which this assumption fails.

Assuming there is a chunk currently executing on device i, if U_i is the number of work groups in the uncompleted chunk and T_i is the elapsed time since that chunk began execution, we can estimate the remaining time before the chunk completes, λ_i, as $\lambda_i = \Omega_i U_i - T_i$. If there are no chunks currently executing, then $\lambda_i = 0$.

Let W be the total number of work groups remaining to be scheduled and W_i the optimal number of work groups to schedule on device i. Assuming that we schedule all of the remaining work, then $W = W_1 + W_2$; solving for W_2 yields:

$$W_2 = W - W_1 \tag{4.1}$$

To minimize execution time, we want both devices to complete at the same time, satisfying the following[3]:

$$\Omega_1 W_1 + \lambda_1 = \Omega_2 W_2 + \lambda_2 \tag{4.2}$$

Substituting Equation 4.1 into Equation 4.2 and solving for W_1 yields:

$$W_1 = \frac{\lambda_2 - \lambda_1 + \Omega_2 W}{\Omega_1 + \Omega_2} \tag{4.3}$$

[2] For performance reasons, the minimum scheduling granularity is actually much larger than an individual work group. To simplify the discussion, we compute the optimal partition in terms of work groups.

[3] To extend this algorithm to $N > 2$ devices, this single equality would be expanded into a system of $N - 1$ equalities.

The number of work groups must be an integer, so instead of W_1 we must use $\lceil W_1 \rceil$ or $\lfloor W_1 \rfloor$. In extreme cases, the difference in performance between the two partitions may be large, so we compute the expected completion time for both and use whichever we expect to finish earlier.

4.3.2 Scheduling Algorithm

We now present the complete scheduling algorithm, parameterized by the variables shown in Table 4.2:

1. Launch one chunk of size β on each device.
2. When a chunk completes execution on device D:
   (a) If all devices have completed at least γ chunks, proceed to step 3.
   (b) Otherwise, launch another chunk on D, increasing the chunk size by a factor of δ; return to step 2.
3. Partition the remaining work using Equations 4.1 and 4.3, sending W_1 work groups to device 1 and W_2 work groups to device 2.

The goal of the scheduling algorithm is two-fold: to determine quickly and accurately the relative performance rates in cases when devices provide similar levels of performance, and allow the faster device to execute a large amount of work (or even all of the work) without waiting for the slower device in cases when the devices are significantly imbalanced. The algorithm begins with relatively small chunks to address the former concern but exponentially increases the chunk size to address the latter concern. For both cases, we would like to use a small number of chunks, because, as we saw in the previous chapter, the overhead of scheduling many small data transfers and kernel invocations would be prohibitive.

For the results presented in this chapter, we set the initial chunk size, β, to 7% of the total work and the chunk growth rate, δ, to 1.5x; in Section 4.5.4, we explore the impact of

	Parameter	Value
β	Initial chunk size (fraction of total work)	7%
γ	Minimum completed chunks per device	2
δ	Chunk growth rate	1.5x

Table 4.2: Load-balancing algorithm parameters.

the initial chunk size and the growth rate on the overall performance. We set the minimum number of completed chunks per device, γ, to 2 because the performance of the first chunk is often slightly worse than later chunks and is thus a less accurate predictor of expected performance.

Current GPUs are non-preemptive, so a kernel cannot begin execution until all previously scheduled kernels have completed. Thus, any scheduling algorithm that blindly sends a fixed amount of work to all available devices may wait an unbounded amount of time for that work to complete. A lack of observed forward progress may be due to a number of different causes: starvation caused by another kernel, temporary or permanent unresponsiveness due to software or hardware failures, or a severe performance anomaly. To deal with all of these scenarios effectively, we extend our algorithm slightly. In the second step of the algorithm, if device A claims all of the remaining work and device B has not yet completed a single chunk, we send all of the uncompleted work (including the work already sent to device B) to device A and no longer wait for device B to complete its work[4]. We explore the effectiveness of this approach in Section 4.5.6.

4.3.3 Scheduler Implementation

The previous discussion described the scheduling algorithm at an abstract level. Now we describe at a more practical level what it means to schedule a chunk on a device.

[4]In this context, it would be useful to have a mechanism by which an enqueued OpenCL operation that has not begun execution could be canceled. In the absence of such functionality, we simply "forget" about the unwanted operations; they may execute at some point in the future but the application's progress is no longer gated by their completion.

Before a set of work groups in a chunk can begin execution on a device, we must ensure that the input data needed by the work groups is present in the device's memory. Similarly, after execution completes, we must ensure that the output data generated by the work groups is copied back to host memory (assuming that the data is consumed by the host program). When the scheduler decides to assign a chunk to a given device, it first determines the set of input and output data consumed and produced by the chunk (this can be accomplished by analyzing the kernel source code as described by Kim et al. [50]). The scheduler then enqueues commands to copy any required input data, launch the relevant subset of the kernel, and copy any generated output data. A callback is registered on the final operation in the chunk so that the scheduler is re-invoked once that operation completes.

4.3.4 Example Schedule

Figure 4.4 shows a real example of the sequence of operations scheduled by the dynamic load balancer while running DCT [4]. Here we describe, in chronological order, the steps taken by the scheduler:

1. The scheduler begins by sending an initial chunk to each device, each representing 7% of the total work groups available. Each chunk comprises three operations: transferring input data to the device, invoking a subset of the kernel, and transferring output data back to host memory.

2. Because the discrete GPU provides higher performance on this application, it completes its first chunk earliest, at 16 milliseconds. The scheduler sends it a new chunk that is 1.5x as large as the initial chunk.

3. The integrated GPU completes its first chunk at 23 milliseconds. The scheduler sends a new, larger chunk to the integrated GPU.

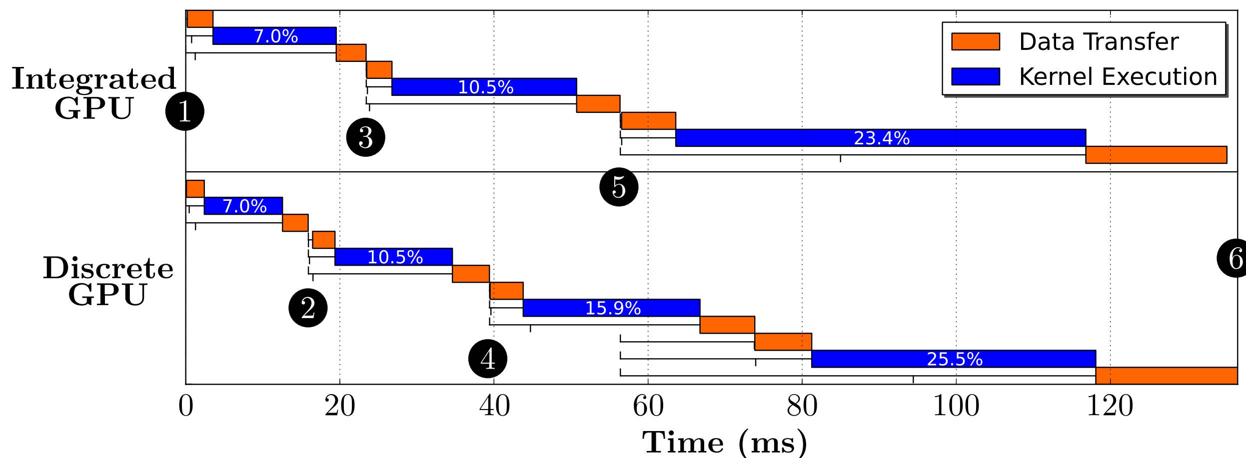


Figure 4.4: Example schedule generated by the dynamic load balancer for the application DCT. The left and right edges of a box represent the start and end time, respectively, of a given operation. Upticks indicate the time an operation was enqueued by the scheduler; downticks represent the time an operation was submitted to a device by the OpenCL runtime. Each kernel execution is labeled with the percentage of total work groups scheduled in that chunk.

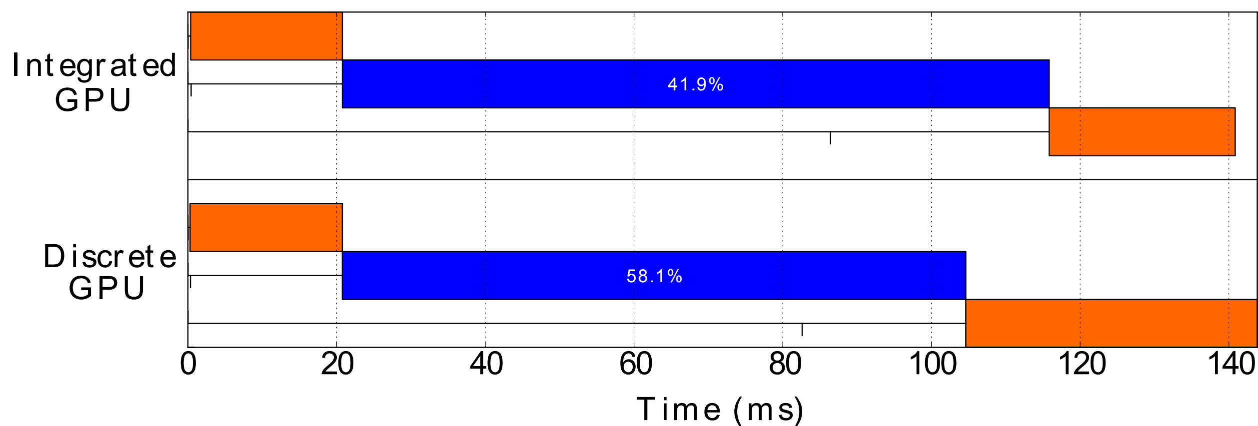


Figure 4.5: Optimal static schedule for DCT, discovered via an exhaustive search of all partitions.

4. The discrete GPU completes its second chunk at 39 milliseconds. Because the integrated GPU has not yet completed its second chunk, the scheduler sends a third, even larger chunk to the discrete GPU.

5. The integrated GPU completes execution of its second chunk at approximately 56 milliseconds. Both devices have now finished two chunks and the scheduler has enough information to schedule the remaining work. It took the integrated and discrete GPUs an average of 51 and 35 microseconds, respectively, to finish each work group (including data transfers and kernel execution) in the most recently completed chunk. Based on this information alone, we would conclude that we should schedule 59% of the remaining work on the discrete GPU. But this ignores the time required to complete the chunk that the discrete GPU is already executing. The scheduler estimates that this chunk will take 34 milliseconds in total; because 17 milliseconds have already elapsed since that chunk began execution, the scheduler estimates that the chunk will take another 17 milliseconds to complete. Using Equation 4.3, the scheduler decides to send 47% of the remaining work (23% of the total work) to the integrated GPU and the rest to the discrete GPU.

6. This scheduling decision proves to be nearly optimal: both devices complete execution of their final chunks about 1 millisecond apart, at around 136 milliseconds.

For comparison, Figure 4.5 shows the equivalent static schedule for the same application.

4.4 Experimental Setup

We characterized the performance of our proposed load-balancing approach using the six OpenCL applications shown in Table 4.3. Five of the applications are from version 2.7 of the AMD Accelerated Parallel Processing (APP) SDK [4] and one (K-Means) is from version 2.1 of the Rodinia benchmark suite [22]. Each application executes a single kernel at a time,

Application	Data Set
Black-Scholes	12.8M samples
Discrete Cosine Transform (DCT)	6K x 6K matrix
Fast Fourier Transform (FFT)	32K 1K-element vectors
Matrix Multiplication	4K x 4K matrices
Mersenne Twister	29M random numbers
K-Means	800K 34-dimensional points

Table 4.3: Benchmarks and data sets used for evaluation.

and the host program consumes the output of the kernel.[5] For each application, we chose a data-set size close to the maximum size supported by our system. The original version of FFT supports the processing of only a single vector of 1K elements. To achieve more reasonable execution times, we modified FFT to support an arbitrary number of vectors.

Although we envision our proposed load-balancing technique being applied in an automated fashion, as suggested by prior work [50], for this study we manually modified each application to add support for load balancing. We measured all performance results using the same version of the application, which can execute natively on a single device or load-balance using our dynamic scheduling algorithm, the three self-scheduling algorithms discussed earlier, or a fixed partition.

We define the execution time for a single run as the total time required to transfer input data to the GPU(s), complete execution of the entire kernel, and transfer output data back to host memory. Unless otherwise stated, all results represent the average (arithmetic mean) across 25 runs, with 2 preliminary runs ignored to avoid initialization overheads. Because K-Means inherently requires multiple kernel invocations, we first averaged its performance across all of the invocations for a single run and then averaged across 25 separate runs; the data set we used converges after 20 invocations.

We measured all performance results on a system with an AMD A8-3850 APU (a 2.9-GHz quad-core CPU and an integrated AMD Radeon HD 6550D GPU) and a discrete AMD Radeon

[5]K-Means requires multiple invocations of its kernel to converge on a solution, but the output of one kernel invocation is not directly consumed by the next invocation.

HD 6670 GPU; we use only the two GPUs for kernel execution. We compiled the benchmarks with Microsoft Visual Studio 2010 version 10.0.303191.1 and executed them in Windows 7 with AMD Catalyst version 12.8. All benchmarks use single-precision floating-point arithmetic. Unless otherwise stated, we ran both GPUs at their default frequencies.

4.4.1 Filtering Performance Anomalies

In the course of our measurements, we frequently observed performance anomalies that caused unexplained slowdowns of up to an order of magnitude or more for data transfers and, less frequently, kernel execution. These anomalies occurred most frequently with the dynamic scheduler, but also occurred repeatedly with the static scheduler and even native execution. Of course, some performance degradation due to load balancing is expected, especially for concurrent data transfers that compete for the host memory system. However, we do not believe these anomalies were caused by contention for three reasons. First, the slowdowns were significantly more severe than the slowdowns that would be expected (and that we observed) from memory contention. Second, the anomalies *never* occurred in FFT, which is the most transfer-bound application we studied and thus the one in which we would most expect contention to matter. Third, as noted earlier, the anomalies occurred even during native execution of some applications, when contention from another device does not occur. Our investigations strongly suggested that these slowdowns were due to inefficient interactions among the operating system, graphics driver, and OpenCL runtime.

To ensure that our results reflect differences due to scheduling strategies rather than issues with constantly evolving systems software, we ignored any runs in which these performance anomalies occurred. To determine objectively when such an event occurred, we first computed the transfer and compute throughput[6] of each device for each run with a given configuration. We then determined the best transfer and compute throughput for each device across all

[6] We compare throughputs instead of execution times because the dynamic scheduler may send different amounts of work to a given device on different runs, and thus we would expect changes in execution time even in the absence of fluctuations in the underlying performance.

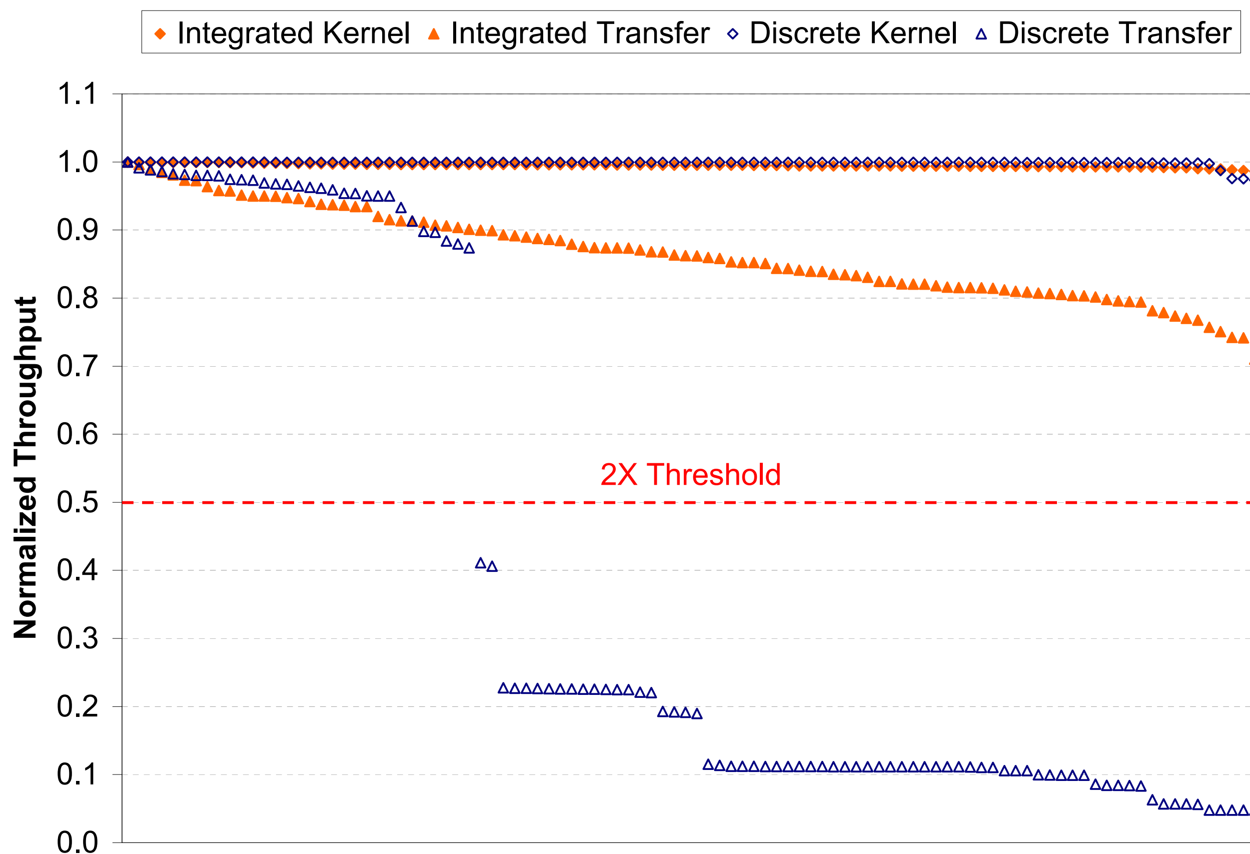


Figure 4.6: Normalized throughput of kernel execution and data transfer for the integrated and discrete GPUs during 100 runs of dynamic load balancing of Black-Scholes. Each of the four metrics is independently sorted in descending order from left to right; thus, points at the same location on the X-axis may not correspond to the same run. Using a threshold of 2x, runs in which at least one of the four metrics falls below 0.5 would be thrown out.

runs with a given configuration and discarded any runs in which the compute or transfer throughput of a device was more than 2x worse than in the best case. We applied this filtering consistently, regardless of whether we were measuring the performance of dynamic or static load balancing or native execution.

Figure 4.6 shows the distribution of kernel and transfer throughputs for 100 separate runs of the worst case: Black-Scholes using dynamic load balancing. The distribution of the poorly performing metrics is clearly bimodal. The upper-left cluster most likely represents fundamental performance losses due to contention or other load balancing-related overheads, while the lower-right cluster represents the performance anomalies we wish to filter out. For all configurations and applications, including those not shown here, a threshold of 2x reliably

Application	Native Execution		Load Balancing	
	Integrated	**Discrete**	**Static**	**Dynamic**
Black-Scholes	0	0	0	69
DCT	0	0	0	0
FFT	0	0	0	0
K-Means	0	0	0	18
Matrix Mult.	0	0	1	32
Mersenne	35	0	37	35

Table 4.4: Of 100 runs, number of runs discarded using a threshold of 2x.

divides these two clusters; however, some significant performance reductions may *not* be filtered out. For example, the transfer throughput to the integrated GPU in Black-Scholes drops by up to 30% during dynamic load balancing, but any runs with such poor performance would be filtered out only if one of the three other metrics were below the threshold.

Table 4.4 shows, out of 100 runs, the number of runs for each application and configuration discarded using a 2x threshold. For DCT and FFT, no results were filtered out, while for Black-Scholes, K-Means, and Matrix Multiplication, runs were filtered out only for dynamic load balancing (with the exception of a single anomalous run for the static load balancer on Matrix Multiplication). For Mersenne Twister, all configurations except native execution on the discrete GPU were affected essentially equally by filtering. To ensure that we always had 25 runs across which to average, we collected data for more than 25 runs but used only the first 25 runs that remained after filtering.

4.5 Results

We measured the effectiveness of our proposed dynamic approach to load balancing by comparing it against the optimal fixed partitions[7] in two different cases, in which the performance of the underlying devices remains fixed or varies. We also compared our dynamic

[7] More precisely, we mean the optimal fixed partition *that schedules at most one chunk on each device*. It is possible that using multiple chunks per device could slightly improve performance in some cases, although it is more likely to hurt performance if applied indiscriminately. This consideration is outside the scope of this chapter, and notably has not been considered by prior work that uses fixed partitions.

approach to the self-scheduling approaches identified earlier. We then characterized the quality of the dynamic scheduler's prediction as well as the scheduler's sensitivity to both the algorithmic parameters discussed earlier and the data-set size. Finally, we evaluated the scheduling algorithm's ability to respond to extreme performance imbalances.

4.5.1 Load Balancing without Variability

We first present results measured with no performance variability; that is, with the performance of each device fixed. We compare our proposed dynamic approach to the best fixed work partitioning, discovered via an exhaustive search of all partitions from 0% to 100% in increments of 1%. This represents an upper bound on the performance of static load balancing.

Comparison to Static Scheduling

Figure 4.7 shows the overall speedup of both dynamic and static load balancing relative to single-device execution on the fastest device in the system (the discrete GPU). Overall, the dynamic scheduler was only 2.2% slower on average than the static optimal across all six applications. The dynamic scheduler provided an average speedup of 1.48x relative to native execution, compared to the optimal fixed partition's average speedup of 1.51x. The dynamic scheduler's performance on individual applications fell into three categories:

Faster: The dynamic approach was 6.6% faster on DCT due to better transfer performance. In a fixed partition, the data transfers occur only at the beginning and end of execution, meaning that the data transfers to and from the two devices are likely to occur concurrently and thus be slowed by contention. The dynamic scheduler, on the other hand, spreads out the data transfers, leading to less contention. This difference in behavior is readily apparent in Figures 4.4 and 4.5. The two devices transferred data simultaneously 73% of the time for the fixed partition but only 25% of the time for the dynamic scheduler.

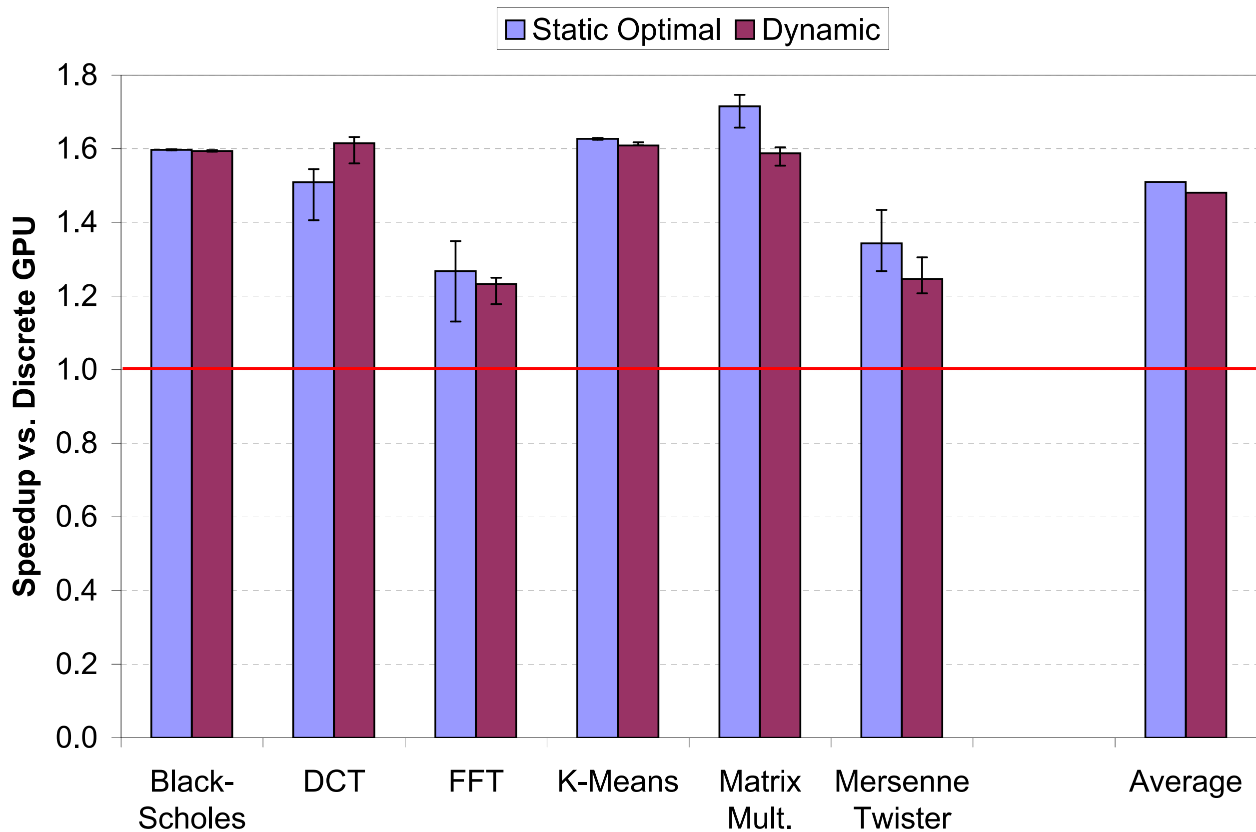


Figure 4.7: Speedup of dynamic and static load balancing relative to single-device execution on the discrete GPU. The static load balancer uses the optimal partition, discovered via exhaustive search. Error bars show the minimum and maximum speedups observed.

Same: The dynamic approach was marginally slower on Black-Scholes and K-Means, by 0.2% and 1.1%, respectively.

Slower: The dynamic approach was slower on FFT, Matrix Multiplication, and Mersenne Twister, by an average of 6.2%. FFT and Mersenne Twister are highly transfer-bound, which limits the benefits of load balancing in general: these two applications achieved the lowest speedups in both the static and dynamic cases. Transfers to the discrete GPU were significantly slower in FFT and Mersenne Twister for the dynamic scheduler than for the fixed partitions (by 17% and 20%, respectively) because transfer performance suffers more than kernel performance when breaking an operation across multiple chunks. Additionally, for all three of these applications, particularly Matrix Multiplication, the performance of

Application	**Sequential Training**	**Concurrent Training**
Black-Scholes	1,806	887
DCT	-	-
FFT	63	19
K-Means	275	132
Matrix Multiplication	37	16
Mersenne Twister	26	9
Overall (Actual)	738	330
Overall (Normalized)	702	300

Table 4.5: Number of times each kernel must be run before the static approach can overcome its training overhead and reduce the total execution time relative the dynamic scheduler. We consider two hypothetical approaches to training: *Sequential* executes a kernel natively on each device, one after the other; *Concurrent* executes a kernel natively on each device at the same time.

larger chunks is difficult to predict from the performance of smaller chunks, which led to less efficient dynamic partitioning. This issue is discussed further in Section 4.5.3.

Training Overhead

The raw performance data alone does not tell the whole story. One of the principal advantages of our proposed dynamic scheduler is that it requires no training. A static scheduler, on the other hand, must be trained the first time a given application is executed. For the results shown here, we trained the static scheduler using an exhaustive search of all partitions to provide an upper limit on the performance of static load balancing. In practice, less costly training methods would be used, which may result in less optimal fixed partitions. Because of this training overhead, even when static load balancing is faster than dynamic load balancing, the static approach may require many runs of the same application before it can overcome its initial training overhead to provide a lower overall execution time.

Table 4.5 shows the number of runs of each kernel that would be required for the optimal fixed partition to outperform the dynamic approach if we take into account training overhead. We consider two hypothetical training strategies, in which the complete kernel (including

requisite data transfers) is either run sequentially or concurrently[8] on the two devices. We conservatively assume that both training strategies are able to find the same fixed partition found by the exhaustive search (i.e., the optimal partition). Because the dynamic scheduler provides better performance on DCT, there is no point at which the static scheduler breaks even.

The second to last row in Table 4.5 shows the total number of kernel executions that would be required for the optimal fixed partitions to outperform the dynamic scheduler if we assume that all six kernels are run the same number of times. Because the different kernels have execution times that differ by as much as 7x, the overall result is heavily weighted by the performance on the two longest-running kernels, Black-Scholes and Matrix Multiplication. To address this, the last row shows how many total kernel executions would be required if all six kernels had the same (statically partitioned) execution time. In both cases, static load balancing makes sense only when we are sure that we will run applications hundreds of times.

Comparison to Self-scheduling

Figure 4.8 shows the performance loss, relative to our dynamic scheduler, of the three dynamic self-scheduling algorithms described in Section 4.2: chunk self-scheduling (CSS) [51], guided self-scheduling (GSS) [80], and trapezoid self-scheduling (TSS) [104]. For all three algorithms, we swept either the chunk size (CSS) or the *minimum* chunk size (GSS and TSS) from 1% to 25% of the total work, in increments of 1%, and report results only for the best parameter value. Relative to our proposed algorithm, CSS with a chunk size of 21% was on average 3.4% slower, GSS with a minimum chunk size of 22% was 11.2% slower, and TSS with a minimum chunk size of 4% was 2.7% slower. Artificially increasing the number of processors in the GSS algorithm (from two to four) to decrease the chunk size, as suggested by Tzen

[8] We conservatively assume that executing on both devices concurrently does not cause contention. In practice, contention will slow both devices, resulting in a larger training overhead. Note also that concurrent training may make the slower device appear faster than it will actually be during load balancing, because during the end of its execution there will be no contention from the already-completed faster device. Thus, concurrent training will most likely result in less optimal fixed partitions than sequential training.

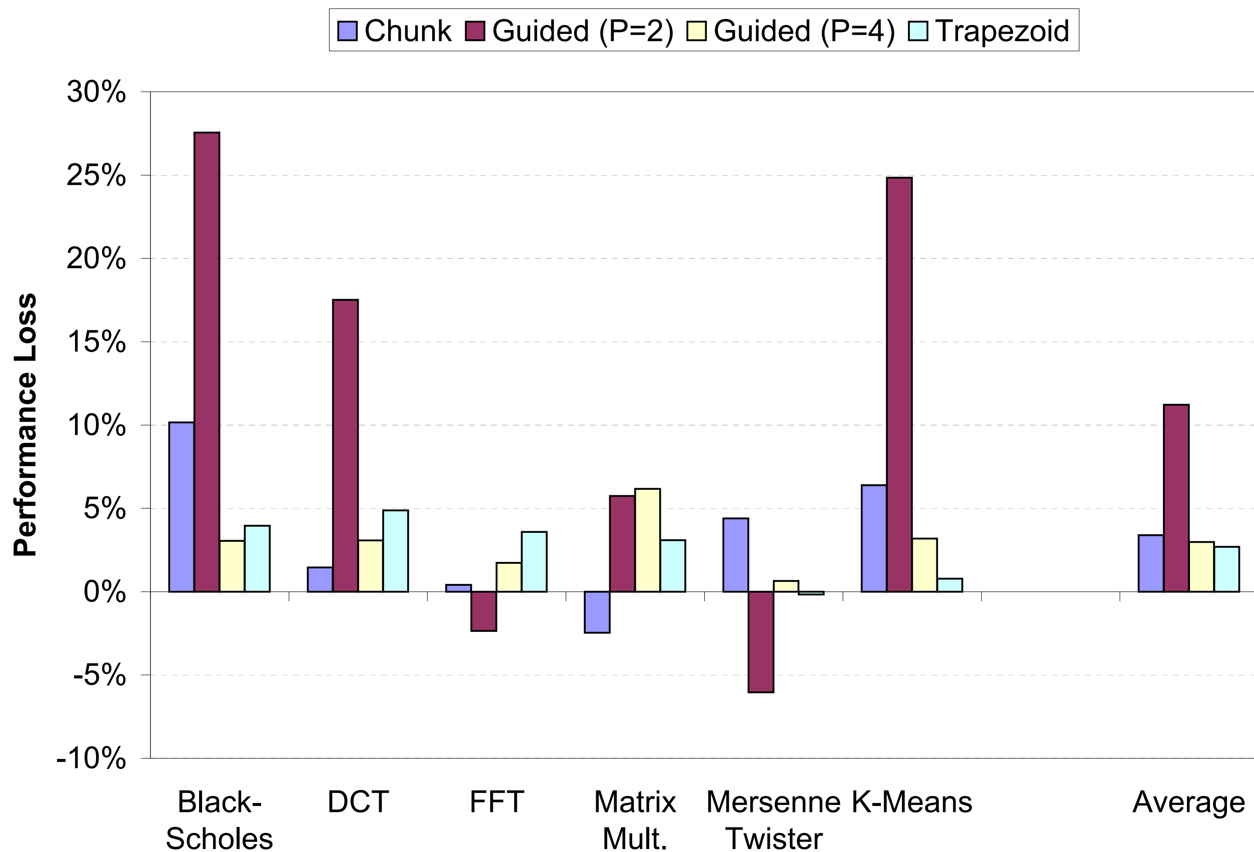


Figure 4.8: Performance loss of self-scheduling algorithms relative to our dynamic algorithm.

and Ni [104], improved the average slowdown of GSS to 3.0% (with a minimum chunk size of 16%).

4.5.2 Load Balancing with Variability

To measure the impact of performance variability, we varied the discrete GPU's core clock frequency from its nominal value of 800 MHz down to a minimum of 200 MHz, in increments of 100 MHz. Although frequency scaling itself may be an important source of performance variability, it can also be considered a proxy for other sources of performance variability, such as contention. We used frequency scaling in these experiments because it easily controllable and repeatable.

We first discovered, via exhaustive search, the optimal fixed partition for each application at each frequency. We then measured the performance of these fixed partitions as well as

the dynamic scheduler across the entire range of frequencies. We used as a baseline a static oracle that always selects the optimal fixed partition at each frequency.

Average Performance

Figure 4.9 shows the execution time for three fixed partitions and the dynamic scheduler, first normalized to the execution time of the static oracle and then averaged across all frequencies. The first two fixed partitions are those optimized for the minimum and maximum frequencies, respectively; the other partition is the one that provides the best average performance (decided on a per-application basis[9]).

For Black-Scholes, DCT, and K-Means, the dynamic scheduler provided significantly better average performance (14% to 20% better) than even the best fixed partition. The advantage was more modest for Matrix Multiplication and FFT: the dynamic scheduler was 6.3% and 1.2% faster, respectively. The dynamic scheduler performed the worst on Mersenne Twister, where it was 3.9% slower than the best fixed partition. With the exception of Mersenne Twister, the dynamic scheduler was always better on average than the fixed partition optimized for the discrete GPU's nominal frequency (800 MHz). Across all six applications, the dynamic scheduler was on average 9.6% faster than the best fixed partition and 15% faster than the fixed partition optimized for the nominal frequency.

As mentioned earlier, the execution times of both FFT and Mersenne Twister are dominated by the time required to transfer data between host and device memory. Because transfer time is much less sensitive to frequency than is kernel execution, neither application's performance suffered significantly when the frequency was reduced. For both applications, the best static execution time at 200 MHz was only about 16% slower than at 800 MHz, leaving little room for the dynamic scheduler to improve on the fixed partitions.

[9]The best overall partition for Black-Scholes, K-Means, and Matrix Multiplication was the one optimized for 500 MHz; for DCT and FFT, it was the one optimized for 400 MHz; and for Mersenne Twister, it was the one optimized for 300 MHz.

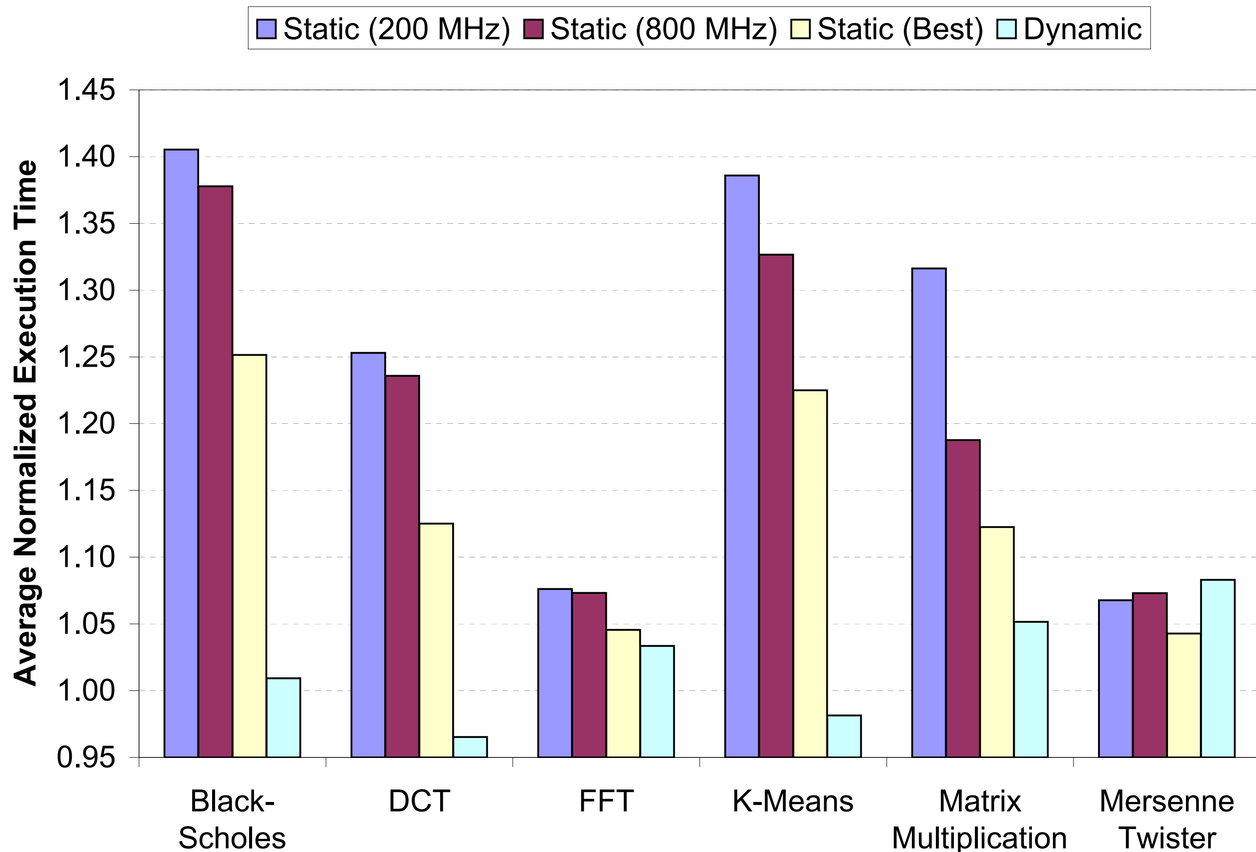


Figure 4.9: Average normalized execution time across a range of discrete GPU core frequencies for three fixed partitions and the dynamic scheduler, relative to the static oracle. The three fixed partitions are the one optimized for the minimum frequency (200 MHz), the one optimized for the nominal frequency (800 MHz), and the one that provides the best average performance.

Across the entire range of frequencies, the dynamic scheduler was never slower than native execution on the fastest device in the system. The same cannot be said for all of the fixed partitions. When the discrete GPU was running at 200 MHz, the fixed partition optimized for the nominal frequency was slower than native execution on the integrated GPU for all six applications. In fact, for four of the applications, at 200 MHz the partitions optimized for 800 MHz down to 400 MHz were all slower than native execution.

Figure 4.10 shows more detailed results for two fixed partitions and the dynamic scheduler for DCT. The fixed partitions are a subset of those shown in Figure 4.3. We can see clearly that while the performance of each fixed partition varied widely over the complete frequency

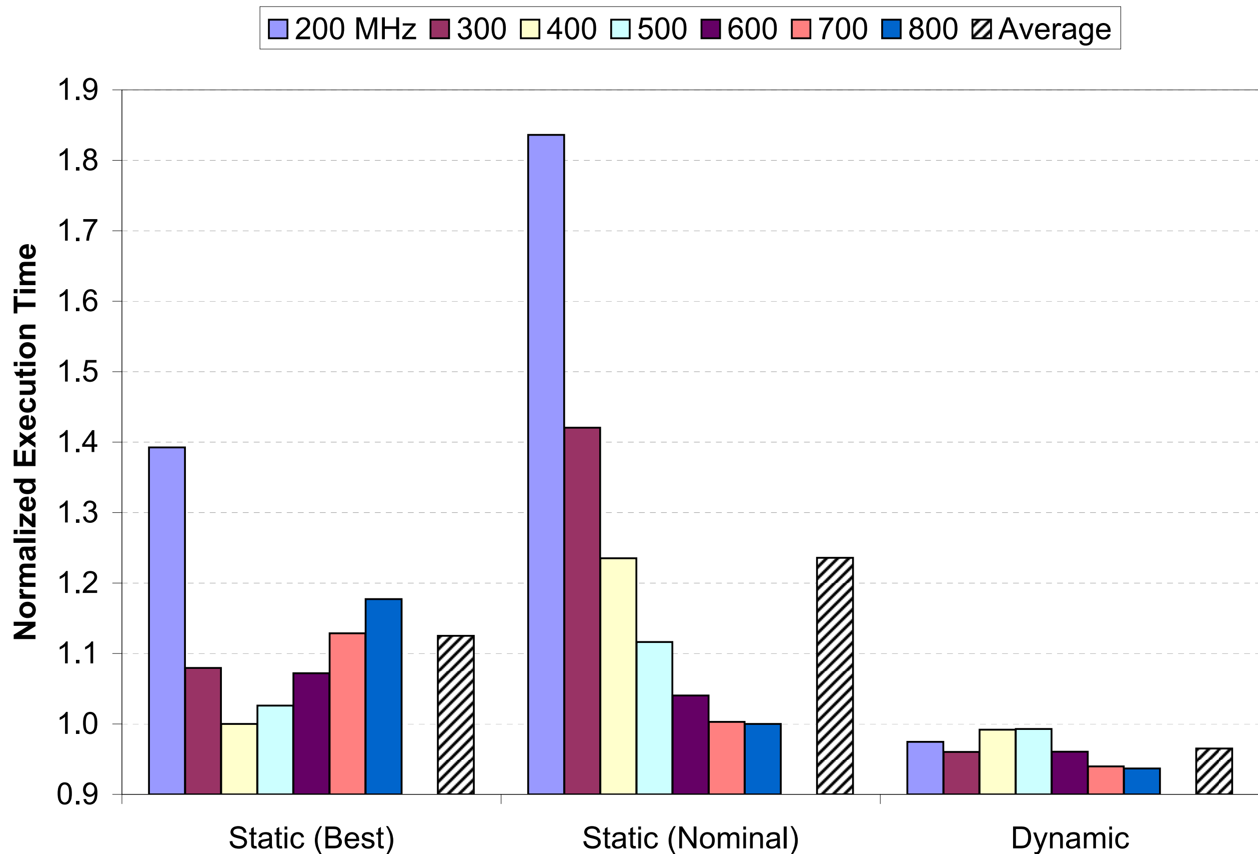


Figure 4.10: Average execution time of DCT for two fixed partitions and the dynamic scheduler, normalized to the best fixed partition at each frequency. The two fixed partitions are the one that provides the best average performance and the one optimized for the nominal frequency (800 MHz).

range, the dynamic scheduler provided a much more consistent level of performance. To achieve good performance with a static approach, we must accurately predict at which frequency the GPU will typically run to choose an appropriate partition. The dynamic approach frees us from this burden because it provides good performance regardless of the specific frequency or, more generally, the relative performance of the underlying devices.

Worst-case Performance

Our analysis so far has focused on average performance. In some scenarios, however, such as when attempting to meet a real-time target or quality-of-service constraint, we may care only about the execution time at whichever clock frequency produces the worst performance.

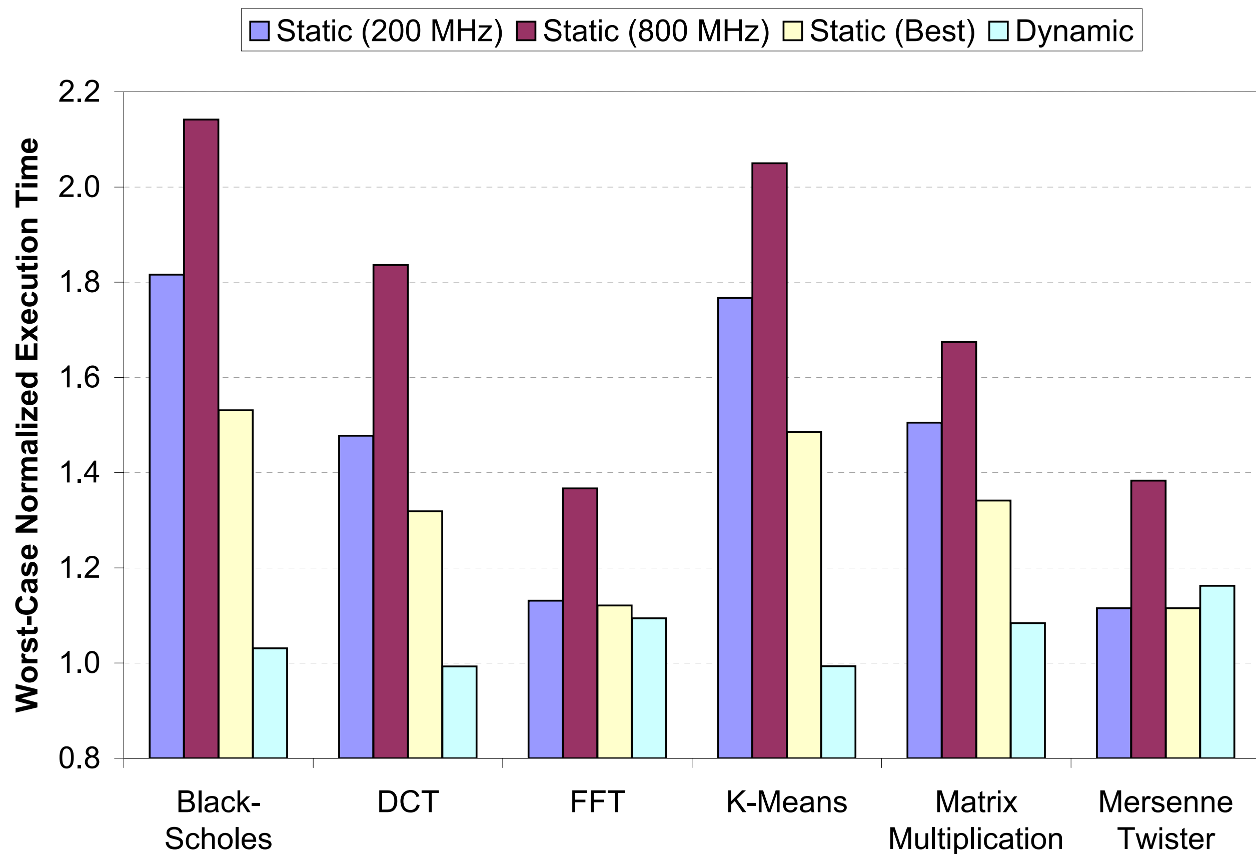


Figure 4.11: Maximum normalized execution time across a range of discrete GPU core frequencies for three fixed partitions and the dynamic scheduler, relative to the static oracle. The three fixed partitions are the one optimized for the minimum frequency (200 MHz), the one optimized for the nominal frequency (800 MHz), and the one that provides the best worst-case performance.

Figure 4.11 shows the highest average normalized execution time for static and dynamic configurations across the entire range of clock frequencies. The best fixed partition in this case is the one with the lowest *maximum* normalized execution time, which for all applications was different from the partition that minimized the *average* time. For four of the applications, the dynamic scheduler provided significantly better worst-case execution time than the best fixed partition (19% to 33% better). Performance was less impressive for the two transfer-dominated applications: the dynamic scheduler's worst-case performance was only 2.4% better than the best fixed partition for FFT and 4.3% *worse* for Mersenne Twister. Overall, the dynamic scheduler was on average 20% faster in the worst case than the best fixed partition. And for

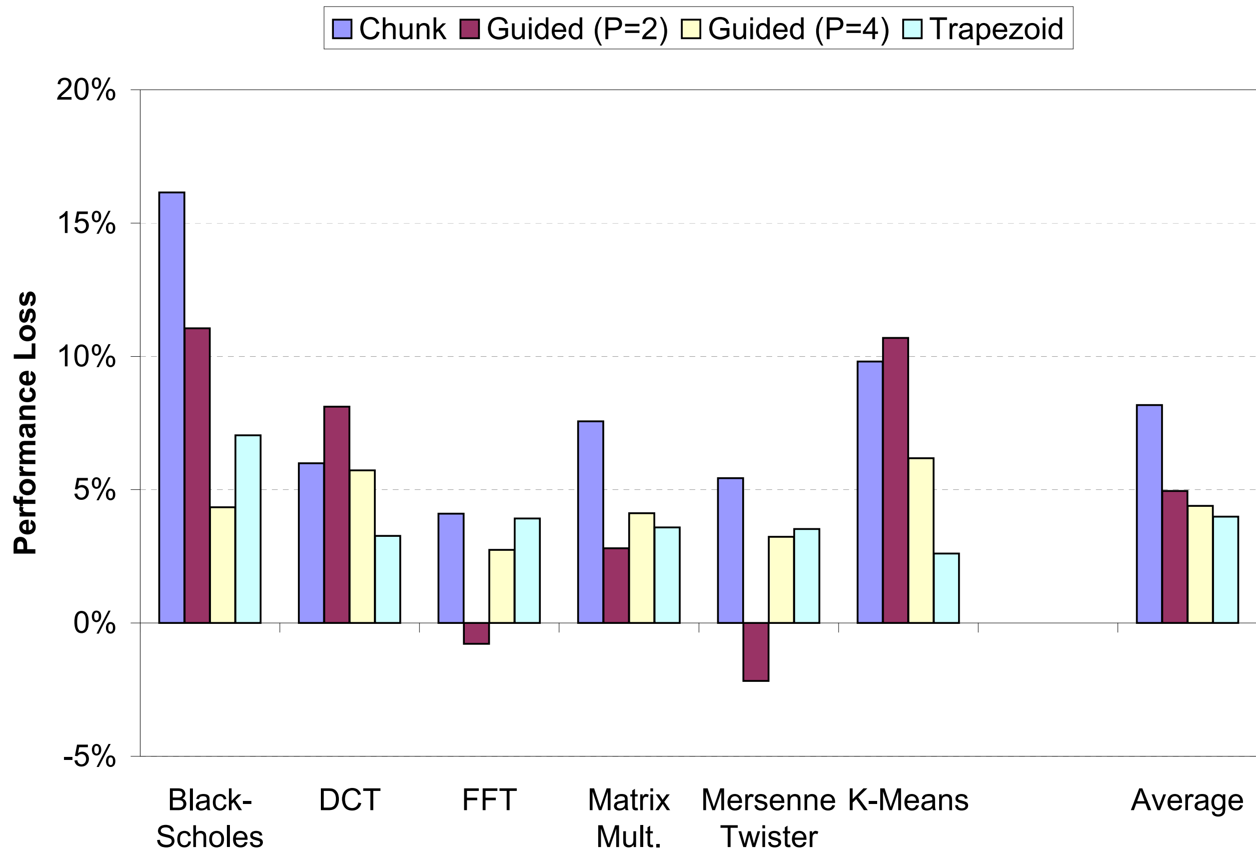


Figure 4.12: Performance loss of self-scheduling algorithms relative to our dynamic algorithm across a range of discrete GPU frequencies.

all six applications, the dynamic scheduler provided better worst-case performance than the fixed partition optimized for the nominal frequency.

Comparison to Self-scheduling

Figure 4.12 shows the performance loss in the presence of frequency scaling, relative to our dynamic scheduler, of the three self-scheduling algorithms. As we did for our dynamic scheduler, we used the same parameter values for the self-scheduling algorithms as we did in the results presented in Section 4.5.1 (i.e., we used the chunk sizes that yielded the best average performance at the default frequencies). Relative to our algorithm, CSS was on average 8.2% slower, GSS was 5.0% slower, and TSS was 4.0% slower. Artificially increasing the number of processors in the GSS algorithm (from two to four) to decrease the initial

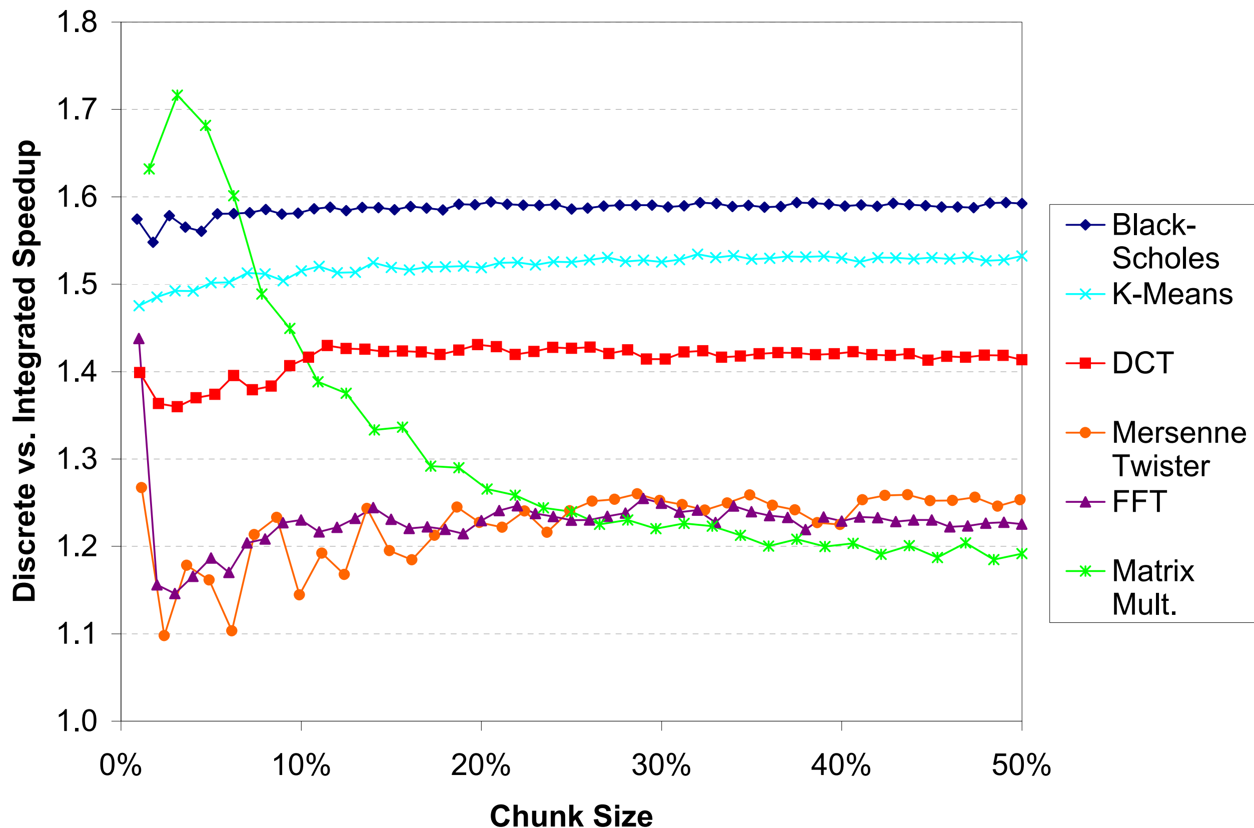


Figure 4.13: Speedup of the discrete GPU relative to the integrated GPU as a function of chunk size.

chunk size significantly increased the performance consistency across the applications and decreased the overall average slowdown to 4.4%.

4.5.3 Prediction Quality

The ability of the load balancer to make efficient scheduling decisions relies on a key assumption: that the relative performance we observe on the small, initial chunks is predictive of the relative performance of the larger, final chunks. To measure how well this assumption holds in practice, Figure 4.13 shows the speedup of the discrete GPU relative to the integrated GPU over a range of chunk sizes.[10] For some applications (Black-Scholes, K-Means, and

[10]Note that this data was measured with the devices executing one at a time rather than concurrently. Thus, the impact of contention between the devices is missing, and the trends we observe here may be slightly different than what the dynamic scheduler must contend with in practice, especially for the transfer-dominated applications (FFT and Mersenne Twister).

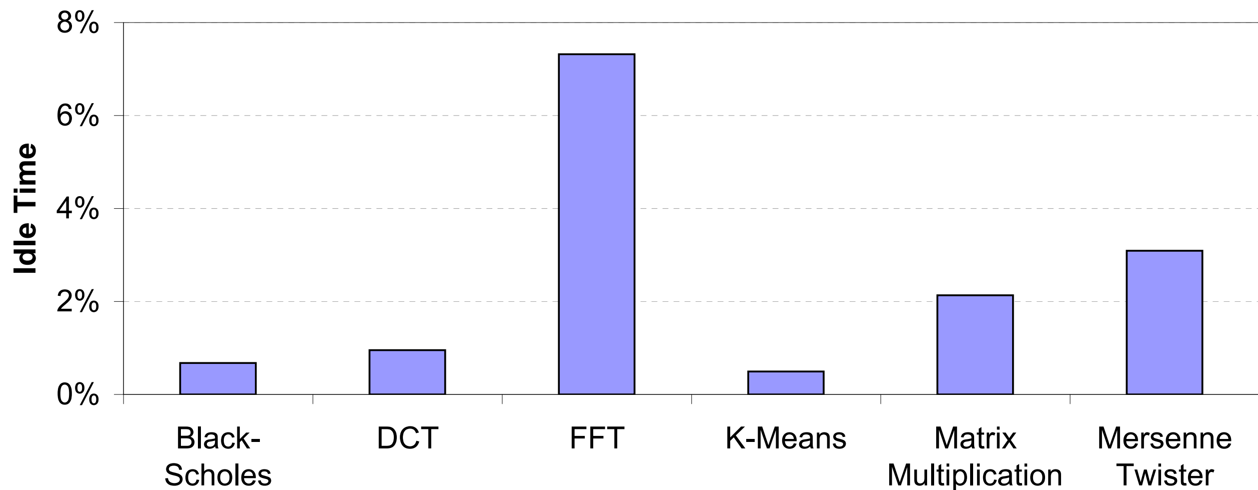


Figure 4.14: Time between the first and second devices finishing execution for the dynamic load balancer, as a fraction of the total execution time.

DCT), the speedup remained relatively constant, and thus we would expect our load balancer to generate efficient schedules. For the other applications (Mersenne Twister, FFT, and Matrix Multiplication), the speedup varied significantly, which we would expect to produce less efficient schedules.

The goal of the dynamic scheduler is for both devices to complete execution at the same time. To demonstrate how close the scheduler came to achieving this goal, Figure 4.14 shows for each application the average time that one device was idle at the end of execution, expressed as a fraction of the application's total execution time. The applications we identified earlier as most predictable (those maintaining consistent speedups across different chunk sizes) had the shortest idle times, averaging 0.7%. The less predictable applications, on the other hand, had longer idle times, averaging 4.2%. This has a direct correlation with performance: the three applications with the lowest idle times also provided the best performance relative to static load balancing.

4.5.4 Sensitivity to Chunk Size and Growth Rate

To determine the optimal parameters for the scheduling algorithm, we measured the performance of each application over a range of initial chunk sizes and chunk growth rates. Note

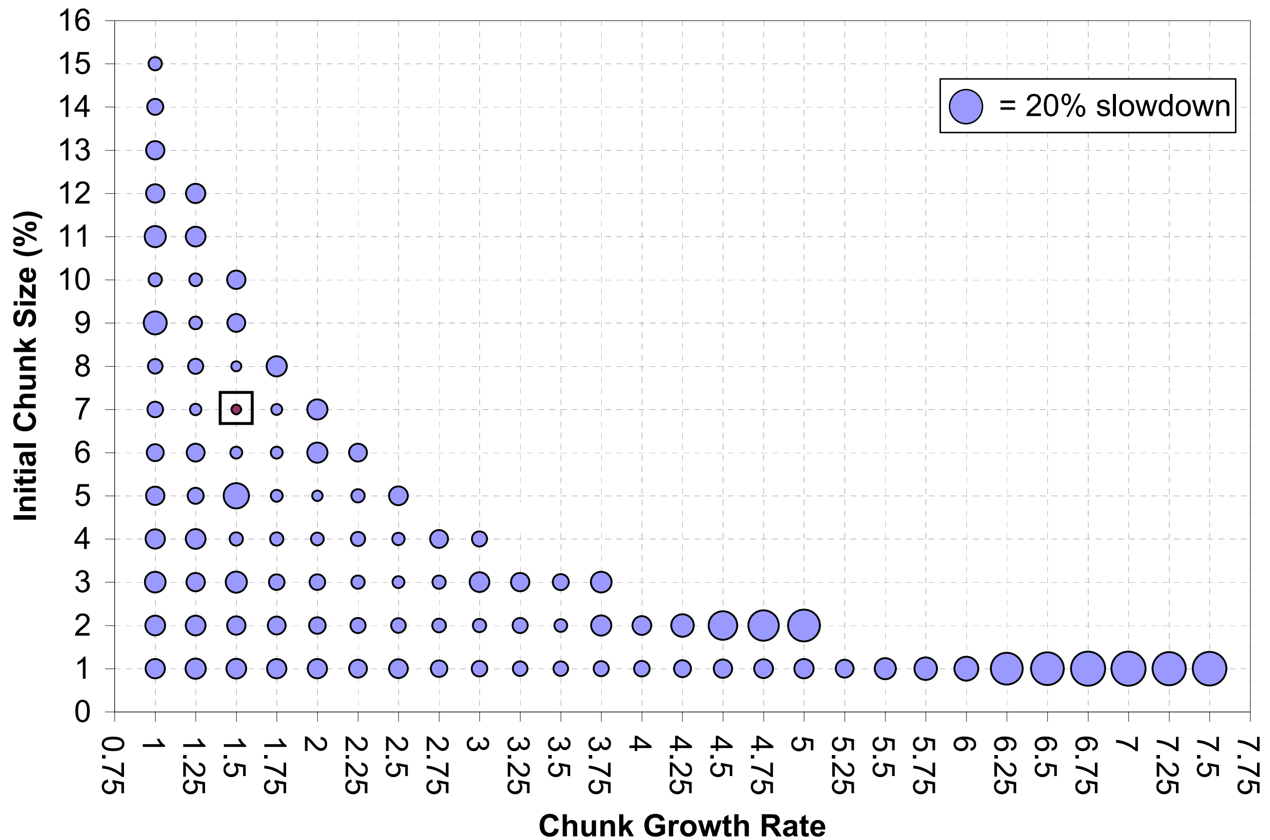


Figure 4.15: Average slowdown across all six applications relative to the optimal parameter values for each individual application. The area of each circle is proportional to the percent slowdown for that combination of chunk size and growth rate. The minimum slowdown, marked with a square, occurs with an initial chunk size of 7% and a growth rate of 1.5x.

that a large chunk size and/or a large growth rate can cause all of work to be assigned to devices in the initial phase of the algorithm, leaving no work for the scheduler to partition in the final phase. To avoid this, we explored all combinations of initial chunk size (in increments of 1%) and chunk growth rate (in increments of 0.25) that resulted in at least 25% of the total work available for the scheduler to partition.[11]

Figure 4.15 shows the performance of each pair of chunk size and growth rate, expressed as the average slowdown across all six applications relative to the optimal configuration for each application. The best overall performance, with an average slowdown of only 1.7%,

[11]We assume that the slower device has completed two initial chunks and the faster device has completed two and started on a third initial chunk before the remaining work is partitioned. This will always be the case when the performance gap between the two devices is no more than 1.5x for the smallest growth rate and 7.6x for the largest growth rate.

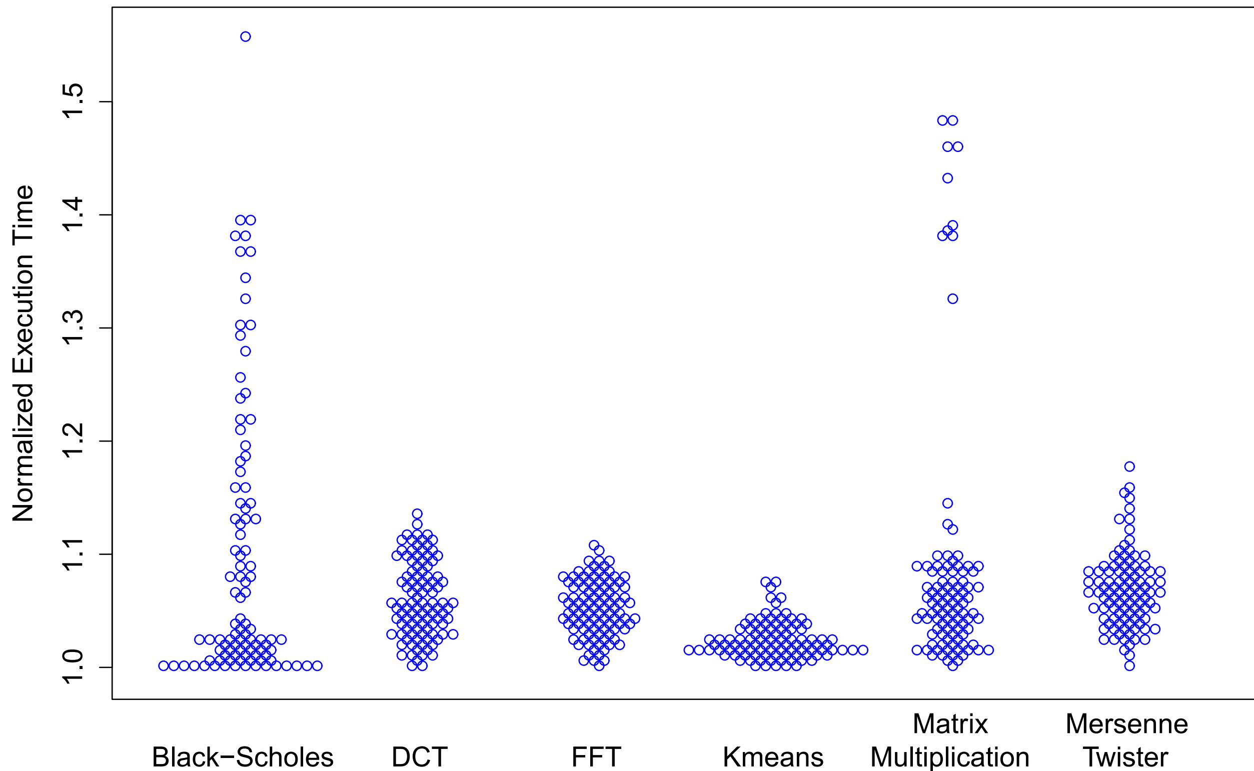


Figure 4.16: Distribution of execution times for each application across the range of initial chunk sizes and chunk growth rates shown in Figure 4.15, normalized to the minimum observed execution time. The horizontal displacement of a data point within a given application has no meaning and is done solely to reduce overlap.

was achieved with an initial chunk size of 7% and a growth rate of 1.5x. Figure 4.16 show the distribution of normalized execution time for each application. Most applications were relatively insensitive to changes in these parameters, with the worst-case slowdown for DCT, FFT, K-means, and Mersenne Twister ranging from 11% to 16%. Matrix Multiplication exhibited similar behavior except for a few outliers. Black-Scholes was the most sensitive, with a worst-case slowdown of 56%.

4.5.5 Sensitivity to Data Size

Our results so far have focused on relatively large data sizes. We now focus on how well the dynamic scheduler performs at smaller data sizes. There are two important effects that we would expect to observe as we scale down the data size. First, for some applications,

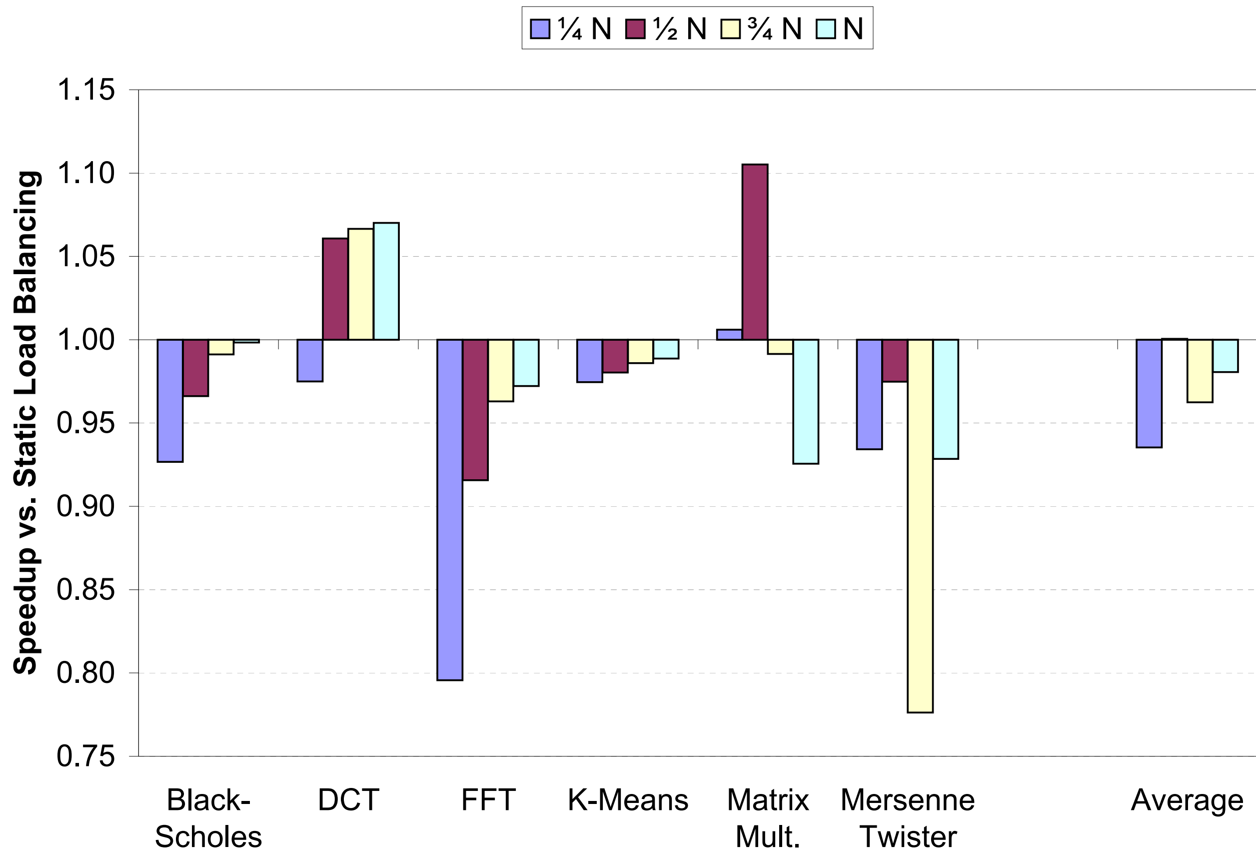


Figure 4.17: Speedup of dynamic load balancing relative to static load balancing for a range of data sizes. Values greater than one indicate that dynamic load balancing is faster than static. N is the default data size specified in Table 4.3.

the optimal partition of work between the two devices will change. This may benefit the dynamic scheduler because it can potentially do a better job of partitioning the work evenly across the two devices. Second, the overhead of using a specific number of chunks remains essentially fixed with decreasing data size even as the total execution time decreases. This means that the *relative* overhead of using more chunks will increase, benefiting the static approach because it uses fewer chunks. Note also that this overhead is more pronounced for data transfers than for kernel executions.

We measured the performance of native execution and load balancing across a range of data sizes. We define the *data size* of an application to be the total amount of data written

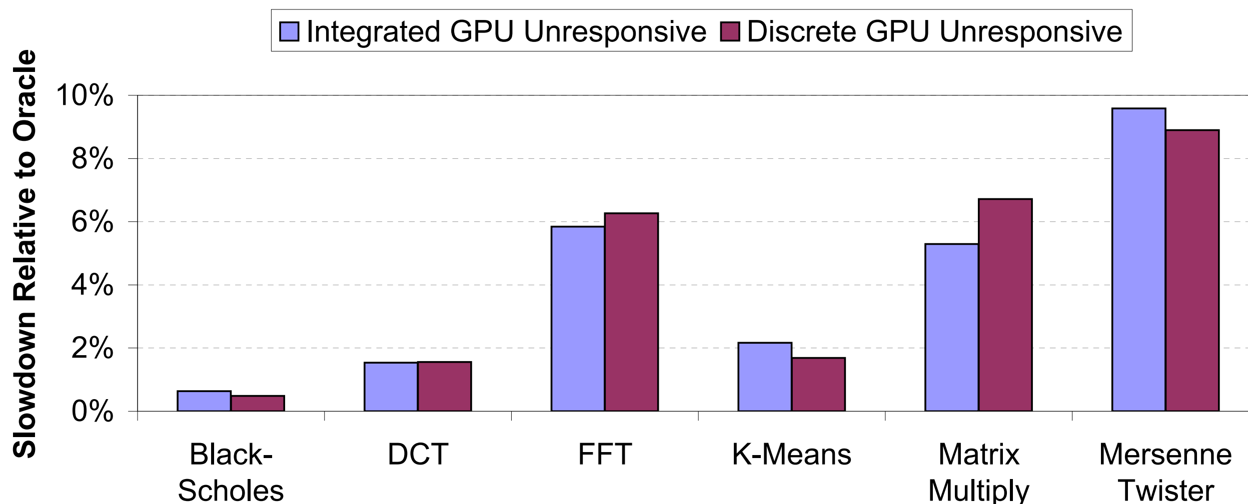


Figure 4.18: Performance of the dynamic load balancer when one device is blocked, normalized to an oracle that sends all work to the unblocked device.

to and read from the GPU during native execution[12]. We set N to the default data size used in previous experiments (and listed in Table 4.3) and swept the data size from N down to $\frac{1}{4}N$ in increments of $\frac{1}{4}N$.

Figure 4.17 shows the speedup of dynamic load balancing relative to static for each data size across all six applications. The dynamic approach performed worst on the two transfer-dominated applications, FFT and Mersenne Twister. This is because the relative overhead of using multiple chunks is larger for data transfers than for kernel execution. Averaging across the four other applications, the dynamic scheduler was actually slightly faster than the static load balancer. Averaging across all six applications, the overall trend was for dynamic load balancing to get slower relative to static load balancing as the data size decreased: dynamic was 2.2% slower at the largest data size but 7.5% slower at the smallest data size. At a data size of $\frac{1}{2}N$, however, the dynamic scheduler essentially matched the performance of static load balancing, with an average slowdown of only 0.3%.

4.5.6 Severe Performance Imbalances

As described earlier, GPUs are non-preemptive and thus an application may wait an unbounded amount of time for a chunk to begin execution on a particular device. We measured the performance of the dynamic load-balancing algorithm when one GPU makes no forward progress; we achieved this by forcing commands sent to that GPU to wait on an event that will never finish. Figure 4.18 shows the normalized execution time of the dynamic load-balancing algorithm relative to an oracle that simply sends all of the work to the unblocked GPU. Any purely static load balancer would be forced to wait arbitrarily long for the blocked device to become free, and would thus become deadlocked in this case. The dynamic approach performed worst on the two transfer-bound applications, FFT and Mersenne Twister, because the use of multiple (five in this particular case) relatively small chunks had a much larger impact on transfer performance than it is does on kernel performance. Overall, the dynamic algorithm was only 3.6% slower on average than the oracle.

4.6 Conclusions and Future Work

Load balancing in heterogeneous systems can provide substantial performance improvements, but only with appropriately chosen work partitions. Existing partitioning approaches require offline training and generate fixed partitions. Using a fixed partition can lead to suboptimal performance as the state of the system or application changes; in some cases, it can lead to worse performance than would be achieved with native execution. To guard against this, we have presented a dynamic load-balancing algorithm that can respond effectively to relative performance changes with no training and with no special knowledge of the source of performance fluctuations. We have demonstrated that our algorithm can provide consistent performance results even in the face of inconsistent system behavior.

[12] We explicitly define data size based on the application's behavior during *native execution* because the amount of data copied to and from the GPU(s) is larger during load balancing for some applications.

In static performance conditions, our dynamic scheduler was only 2.2% slower than the *optimal* fixed partition and still 47% faster than native execution. A real static scheduler, even if it were able to find the optimal partitions, would still require hundreds of separate kernel executions before it would be able to overcome its training overhead. Under dynamic performance conditions, our dynamic scheduler was 9.6% faster than the best fixed partition on average and 20% faster in the worst case. And, unlike fixed partitions, our dynamic scheduler was never slower than native execution, even when the performance of one of the underlying devices changed by a factor of nearly four. Our proposed algorithm can also deal effectively with more extreme scenarios, such as when a device becomes unresponsive. In such a scenario, our scheduler was only 3.6% slower than an oracle that only sends work to the functioning device.

One avenue for future work is to explore a hybrid approach to load balancing that, like a dynamic approach, does not require an offline training phase but, like a static approach, can leverage past performance information for improved partitioning. For example, if Matrix Multiplication were run multiple times, such a hybrid approach might gradually improve its ability to predict the kernel's performance and thereby generate increasingly efficient partitions.

Another obvious direction for future work would be to evaluate our scheduling algorithm on a larger set of applications and hardware systems. Earlier, we hinted at the possibility of load balancing across a CPU and a GPU. Unfortunately, current OpenCL CPU runtime systems provide lackluster performance relative to other CPU programming models (e.g., OpenMP). Improving OpenCL CPU performance is an area of active research [40, 96], so there is hope that CPU-GPU load balancing of OpenCL applications will be an attractive option in the near future.

We initially planned to implement another, even more dynamic scheduling algorithm which involved each device claiming work from a centralized work queue. Some preliminary investigations suggested that this technique might be able to provide even better and more

consistent performance. In addition, this technique should be able to respond effectively to dynamic performance variation that occurs during kernel execution, unlike the algorithm presented earlier. Unfortunately, we discovered that sharing data between kernels on different devices in this way actually violates the OpenCL specification [49] and is not guaranteed to produce correct results. In the future, however, this technique may become viable on platforms supporting the Heterogeneous System Architecture (HSA) standard, which "embraces a fully coherent shared memory model" and allows "developers to write applications that closely couple [CPU] and [GPU] codes" [52].

4.7 Publications

The work presented in this chapter will be published at the Conference on Computing Frontiers in May 2013 [18]. A preliminary version of the work was presented at SRC TECHCON in 2012 [15]. Related work was also presented at SRC TECHCON in 2011 and 2010 [16, 17] and at the AMD Fusion Developer Summit in 2011 [14].

Chapter 5

GPU Frequency Scaling

In previous chapters, we have primarily focused on the benefits of using GPUs, namely their enormous potential to increase performance. In this chapter, we instead focus on one of the potential downsides of using GPUs: their massive power consumption. A high-end GPU can consume as much as twice the power of a high-end CPU, and in many systems the GPU consumes the majority of the total system power. Because of these large power requirements, GPUs represent a ripe target for energy reduction techniques.

One of the most widely used techniques for reducing the energy consumption of a CPU is dynamic voltage and frequency scaling (DVFS) [106], in which the CPU voltage and frequency are increased or decreased in response to workload demands, rather than always running the CPU at its maximum voltage and frequency. Widespread hardware support for frequency scaling in CPUs has led to a large body of work on software techniques for choosing the most energy-efficient frequency. Although GPUs also support frequency scaling, there has been little research on leveraging that capability for energy savings.

An application running on either a CPU or a GPU is typically either compute-bound or memory-bound, meaning that the performance is limited by either the compute cores or the memory system. One useful way to characterize an application as compute- or memory-bound is to consider the rate of memory requests being sent from the cores to the memory. If the

cores are generating memory requests faster than the memory system can complete them, the application is memory-bound. If instead the memory system is completing requests faster than the cores can generate them, the application is compute-bound. Changing the frequency of the cores or memory should change the request or completion rate, respectively.

In a CPU, the frequency of the memory system is typically fixed, so the goal of DVFS for memory-bound applications is to adjust the CPU frequency such that the request rate of the CPU cores matches the completion rate of memory. In a GPU, however, we can control the frequency of both the cores and memory. Thus, the goal of DVFS for GPUs is to determine which component is the bottleneck, and reduce the frequency of the *other* component to match the rate of the slowest component. This extra degree of freedom complicates the rate matching problem, but also allows DVFS to potentially reduce energy for both memory- *and* compute-bound applications.

In this chapter, we show that the only previously proposed DVFS algorithm for GPUs [62] performs poorly on modern GPUs; in fact, it results in higher energy consumption than simply leaving the GPU's frequencies at their default values! To address this problem, we propose a simple heuristic for selecting near-optimal clock frequencies that is based on the observation that good performance and low energy are fundamentally related: in other words, we must maintain high performance to minimize energy consumption. Using actual power measurements from a real system, we show that this heuristic works well across multiple GPU generations.

We focus on using DVFS to minimize the total energy consumption of the system while the GPU is actively executing a kernel, as this is typically the dominant energy cost for applications that leverage GPUs. More specifically, we leverage DVFS to reduce the energy consumption of both the GPU and the CPU during GPU kernel execution. Reducing CPU energy consumption in this way was mentioned briefly in prior work [62] but has not been fully explored nor tested on a real system. We present results for the energy savings of a single system, but our techniques could also be applied to individual nodes of a large-scale

system (e.g., a cluster or supercomputer), which would yield even more substantial energy savings.

The rest of this chapter is organized as follows. In Section 5.1, we provide background information on what DVFS is and how it has been applied to both CPUs and GPUs. In Section 5.2, we describe the workloads and hardware systems we used in our experiments. In Section 5.3, we explore the limits of GPU frequency scaling by determining how much energy can be saved if we have access to actual power measurements. In Section 5.4, we present a novel heuristic for approximating the optimal frequencies without access to power measurements, thereby enabling DVFS to be applied effectively in more realistic conditions. In Section 5.5, we discuss techniques for reducing the CPU energy consumption during GPU computation. In Section 5.6, we discuss related work on reducing the energy consumption of GPUs. Finally, we conclude in Section 5.7.

5.1 Background

In this section, we describe the state of the art in both CPU and GPU frequency scaling.

5.1.1 CPU Frequency Scaling

Dynamic voltage and frequency scaling (DVFS) is a popular technique for reducing CPU power consumption, dating back to as early as 1994 [106]. Because the energy consumption of a processor is typically not proportional to its utilization (i.e., it is not *energy proportional*), it is often beneficial during periods of low CPU utilization to reduce the CPU voltage and frequency from their default levels. Ideally, this reduces power consumption with minimal performance loss. Depending on the situation, the reduction of power may be the primary goal or it may be used as a means of reducing energy consumption. The two goals are related but distinct, because, in the absence of performance improvements, reducing power is a necessary but not sufficient condition for reducing energy. If the overall goal is to save energy,

then power consumption must either be reduced without hurting performance or be reduced by more than execution time is increased.

In principle, the voltage and frequency of a processor can be adjusted independently. Frequency scaling without voltage scaling is much less effective than the other way around, because the dynamic power consumption of an integrated circuit depends only linearly on frequency but quadratically on voltage, and the static power consumption depends quadratically on the voltage but does not depend on the frequency. In practice, the voltage and frequency are typically varied together; each ACPI P-state, for example, typically corresponds to a frequency/voltage pair. This is because increasing the frequency often requires an increase in the voltage to guarantee reliable operation, while reducing the frequency typically means that the voltage can be safely reduced, and not doing so wastes power. In a system where there is a one-to-one mapping between frequencies and voltages, the distinction between setting the frequency and setting the voltage is purely a semantic one. But it is the change in frequency that directly impacts software, and so it is often more useful to think about frequency scaling than voltage scaling. In the remainder of this chapter, we focus on frequency scaling but assume that voltage scales along with frequency, and therefore use the terms *frequency scaling* and *DVFS* interchangeably.

Traditionally, adjusting the processor frequency was an extremely expensive operation as it required rebooting a system and adjusting its BIOS settings. Hardware support for changing the frequency at run time first appeared in low-power embedded and laptop processors but has since migrated to high-performance desktop and server processors, marketed under the brand names AMD PowerNow! or Cool'n'Quiet and Intel SpeedStep or Enhanced SpeedStep. Modern CPUs support a small handful of clock frequencies (e.g., three to ten for a range of AMD and Intel desktop and server CPUs that we surveyed), with the minimum clock frequency typically around half the maximum clock frequency and the other frequencies distributed evenly in-between. The latency of switching between different frequencies is on the order of tens of microseconds [79].

Frequency scaling must be supported by the hardware, but is typically controlled by the operating system or other software. Early approaches to applying DVFS to commodity systems, such as the *performance* and *powersave* governors in Linux [79], were simplistic: the frequency was either set to its maximum value, to ensure maximum performance, or to its minimum value, to improve battery life[1] by reducing power consumption. More advanced approaches have since been developed, such as the *ondemand* governor in Linux [79], which attempts to scale the frequency based on utilization. The ondemand governor is widely used in practice, in part because it is the default governor for popular Linux distributions like Ubuntu.

There are two primary challenges in leveraging DVFS for energy reduction[2]: choosing *when* to change the frequency and choosing *what* frequency to use. Many algorithms have been proposed to address these problems, and Beloglazov et al. [11] provide a good overview of this body of work. The algorithms can be broadly divided into three categories, based on the granularity at which they operate: interval-based, inter-task, and intra-task. Interval-based algorithms use the overall CPU utilization level (or other metrics) during a previous, fixed-length interval to predict the optimal frequency for the next interval; the ondemand governor mentioned earlier is one example. Inter-task algorithms attempt to assign different frequencies to different tasks or processes running on a system, based on static properties derived from the source code or dynamic properties measured using profiling or hardware performance counters. Intra-task algorithms attempt to assign different frequencies to different phases of a single task or process. The majority of inter-task and intra-task algorithms target real-time systems, in which each task must be completed by a specific deadline and both the deadlines and set of tasks typically are known a priori. The goal in such systems is to minimize energy without missing any deadlines, and there is no direct benefit to improving performance beyond what is needed to satisfy the deadlines.

[1] Note that improving battery life and improving energy efficiency are not equivalent. Battery life is a measure of how long the system can run, but not how much work it can complete for a given amount of energy.

[2] Leveraging DVFS for *power* reduction is trivial: simply choose the minimum clock frequency.

A major challenge in automatically adjusting clock frequencies for arbitrary applications is the identification and characterization of distinct phases in the application, which is useful because different phases often have different optimal frequencies. Some researchers have proposed complex dynamic schemes for identifying phases while others have relied on manual identification [43]. Phases in OpenCL applications are, for the most part, explicitly defined by the set of OpenCL API calls made by the application; previous work has exploited a similar property of MPI programs [57, 88, 89] and other parallel programs [56].

5.1.2 GPU Frequency Scaling

Most modern discrete GPUs provide hardware support for DVFS. Unlike CPUs, however, GPUs typically expose two frequencies, one regulating the compute cores and one regulating the memory system.[3] GPUs also offer a larger range of frequencies and much finer control over the frequency, allowing changes in increments as small as a single MHz. As with CPUs, frequency scaling is controlled by software, with two notable exceptions: AMD's PowerTune Technology With Boost and NVIDIA's GPU Boost, implemented in the latest high-end GPUs, allow the hardware to dynamically increase the core clock frequency above the default frequency when sufficient power is available. Similar technologies exist for the CPU, namely AMD's Turbo CORE and Intel's Turbo Boost.

For high-powered discrete GPUs, the ability to adjust frequencies traditionally has been used by consumers to *increase* frequencies from their default values in order to maximize performance. A typical approach is to increase either the core or the memory frequency by some relatively small margin and then test the stability of the system using a stress test. This process continues until the maximum stable frequencies have been found, and occurs completely offline; once the maximum frequencies are found, they are not adjusted dynamically at runtime, except perhaps when the GPU enters an idle state and automatically

[3] In some systems, the frequency of the CPU's memory system can also be adjusted, although typically with one of two important restrictions: either the frequency cannot be adjusted at run time and instead requires a reboot of the system, or the frequency cannot be adjusted independently from the core frequency (i.e., each core frequency has an associated memory frequency).

drops its frequencies to their minimum values. For systems in which maximizing performance is not the primary goal, GPU frequencies may instead be decreased in an effort to decrease power consumption and thereby decrease fan noise (by decreasing temperature), improve stability, or increase battery life. Note, however, that the goal of reducing frequencies is typically *not* to improve energy efficiency.

Prior work has shown that, as we might expect, reducing the memory clock frequency for compute-bound kernels and the core clock frequency for memory-bound kernels has a negligible impact on performance [39] and can thus provide energy savings with little or no performance penalty. Given this capability, an obvious question is how to apply DVFS effectively to a real application running on a GPU. Existing DVFS algorithms for CPUs are not directly applicable to GPUs for three primary reasons. First, most of them target real-time or interactive systems, in which the primary goal is to meet deadlines and ensure responsiveness rather than maximize performance. Second, most CPU algorithms rely on the existence of hardware performance counters that are not available on GPUs. Third, most CPU algorithms need to select only a single frequency, whereas a GPU algorithm must select both a core and a memory frequency. We are aware of no existing CPU DVFS algorithm that does not suffer from at least one of these three limitations.

The DVFS capabilities of modern GPUs have been leveraged in a number of ways in prior work. Lee et al. [55] used frequency scaling to improve the throughput of a GPU, but did not consider the energy implications. Cebrían et al. [21] explored the energy impact of DVFS on GPUs, but did not propose how to achieve energy savings in practice. Liu et al. [59] proposed a DVFS algorithm for real-time applications running on CPU-GPU systems. If sufficient slack exists in the schedule to allow for significantly increasing execution time without missing any deadlines, their algorithm reduces the voltage and frequency of the GPU (or CPU) to consume the slack and thereby save energy. We focus on high-performance computing (HPC) applications, for which significantly increasing execution time is not tolerated and thus no

slack exists; the DVFS algorithm proposed by Liu et al. would be unable to save any energy in such a scenario.

The most relevant prior work is GreenGPU, a GPU DVFS algorithm[4] proposed by Ma et al. [62]. GreenGPU extends an existing CPU DVFS algorithm proposed by Dhiman and Rosing [31], and was evaluated in the context of a single NVIDIA GPU released in 2006. In Section 5.4.4, we evaluate GreenGPU on two modern AMD GPUs and compare its performance to that of our own algorithm.

5.2 Experimental Setup

To study the benefits of applying DVFS to GPUs, we used 24 OpenCL applications, taken from version 2.7 of the AMD Accelerated Parallel Processing (APP) SDK [4] and version 2.1 of the Rodinia benchmark suite [22] and shown in Tables 5.1. Due to limitations of our power measurement infrastructure, we were only able to analyze applications that kept the GPU occupied for tens of seconds. We therefore specifically selected applications that either had this property natively or had kernels that were idempotent and could therefore be run repeatedly without changing the application or kernel's behavior. Both the justification and implementation of this repetition are described in greater detail in Section 5.2.1.

We performed all of our experiments on a machine with a 2.6-GHz, 16-core AMD Opteron 6282 SE CPU, a motherboard supporting PCIe 2, 32 GB of RAM, and a 1.2 kW power supply. The machine ran Ubuntu 10.04.4 with AMD Catalyst 13.1. We compiled the applications with GCC 4.4.3. Unless otherwise noted, CPU frequency scaling was controlled automatically by the default governor (ondemand) running at its default settings.

We used two different GPUs in our experiments, an AMD Radeon HD 5870 and an AMD Radeon HD 7970. The HD 5870 was AMD's highest performing single-GPU graphics card

[4]GreenGPU also incorporates CPU-GPU load balancing (and was discussed briefly in Section 4.2), but here we only consider the DVFS portion of the work.

Application	Data Set	Iterations HD 5870	Iterations HD 7970
AES Decrypt	9K x 9K bitmap	15	60
AES Encrypt	9K x 9K bitmap	15	60
Binary Search	123M elements	10,000	13,000
Binomial Option	2M samples	-	5
Bitonic Sort	16M elements (5870)	-	-
	48M elements (7970)	-	-
Black-Scholes	12.8M samples	1,000	1,500
Black-Scholes DP	64M samples	75	500
Discrete Cosine Transform (DCT)	6K x 6K matrix	60	900
Eigen Value	128K diagonal length	-	50
Floyd-Warshall	6K nodes	-	-
Fluid Simulation 2D	2K x 2K grid	500	1,250
Gaussian Noise	11K x 11K bitmap	250	600
Gaussian Noise GL	11K x 11K bitmap	400	600
Histogram	11K x 11K array	750	3,000
LU Decomposition	5K x 5K matrix (5870)	-	-
	7K x 7K matrix (7970)	-	-
Mandelbrot	11K x 11K image	200	75
Matrix Multiplication	4K x 4K matrices	10	20
Matrix Transpose	11K x 11K matrix	750	2,000
Mersenne Twister	29M random numbers	50	500
Monte Carlo Asian	6K x 3K samples	200	250
N-body	128K particles	10	50
Quasi-Random Sequence (QRS)	8K dimensions	10,000	10000

(a) AMD APP SDK benchmarks.

Application	Data Set	Iterations HD 5870	Iterations HD 7970
CFD	200K particles	-	-
Lava MD	32 x 32 x 32 boxes	12	30

(b) Rodinia benchmarks.

Table 5.1: Benchmarks and data sets from the (a) AMD APP SDK [4] and (b) Rodinia benchmark suite [22]. The iterations columns specify the number of times our interposing program repeated the execution of a single kernel.

from its release in 2010 until the release of the HD 7970 in 2012.[5] We also had access to three high-end NVIDIA GPUs, one from each of the last three generations: a Tesla C1060, a Tesla C2050, and a Tesla K20. We originally planned to include these GPUs in our experiments, but NVIDIA has recently disabled frequency scaling under Linux.[6]

To characterize energy efficiency, the most obvious metric to use is the *total energy* consumed by the system during the execution of an application. Ultimately, this is the metric that we want to minimize in the real world. However, it is also worthwhile characterizing how much *extra* energy a system consumes while computing, beyond the energy that it would have consumed over the same time period if it was idle. To avoid biasing our results in the favor of slower, longer-running configurations, we define the *idle energy* for each application as the energy consumed by the idling system over a time period equal to the execution time running at the default frequencies.[7] We then define the *compute energy* to be the total energy consumption of the system minus the idle energy. Although we attempt to state clearly which metric we are using throughout the rest of this chapter, the reader may safely assume that we are using the compute energy except where explicitly stated otherwise. Note that, because of the way we have defined compute energy, if one configuration consumes less compute energy than another, then it necessarily also consumes less total energy.

5.2.1 Measuring Power Consumption and Execution Time

Because energy is the product of power and time, to determine a system's energy consumption while executing a given application, we need to measure the system's average power consumption and the application's execution time. Measuring the former is significantly more challenging than measuring the latter.

[5]The HD 7970 was itself surpassed by the AMD Radeon HD 7970 *GHz Edition*, released later in 2012; the two GPUs are architecturally equivalent but have different default core and memory clock frequencies.

[6]NVIDIA still supports frequency scaling under Windows, but from a high-performance computing (HPC) perspective, this is effectively useless: as of November 2012, only 3 of the top 500 supercomputers in the world ran Windows [102].

[7]Defining the idle energy based on the default execution time makes sense if, once the current application finishes execution, the system will start running another application. We assume this to be the case in most HPC settings.

Techniques for measuring the power consumption of just the GPU have been used effectively in prior work [68, 97]. Unfortunately, these techniques relied on measurement equipment that was unavailable to us. We instead measured the total AC power consumption of the system using a *Watts up? PRO* power meter, which is accurate to plus or minus 1.5% plus 0.3 Watts[8]. Measuring the power in this way had two important limitations. First, we were unable to determine accurately the power or energy consumption of any individual component in the system, including the GPU. Second, due to inefficiencies in the power supply, the AC power consumption of the entire system was greater than the sum of the DC power consumptions of each individual component. In the real world, however, our goal is to reduce the total system energy consumption, including any extra energy consumed by the power supply itself, and so we believe this is a reasonable approach.

Using the same measurement equipment that we used, Ma et al. [62] attempted to more directly measure the power consumption of the GPU: they powered the GPU from one power supply and powered the rest of the system (e.g., the CPU and motherboard) from a second, independent power supply, and measured the power consumption of each power supply separately. Two problems exist with this approach. First, a GPU can draw a non-trivial amount of power (up to 75 W[9]) from the motherboard through the PCIe slot, and this power would be attributed incorrectly to the non-GPU part of the system. Second, power supplies are less efficient at smaller loads, so assuming that a single power supply is replaced with two equivalent power supplies, the total measured power consumption would increase by a potentially non-trivial amount.

The major limitation of the power meter we used is that it measures the instantaneous power consumption only once per second.[10] In order to reduce the effect of random power fluctuations and generate repeatable results, for each experiment we attempted to obtain at

[8]For example, at 100 Watts, the meter is accurate to plus or minus 1.8 Watts.

[9]This is approximately half of the maximum power consumption of the GPU used in their study.

[10]The power meter actually samples the instantaneous power consumption 2,500 times per second. It uses all of the samples when reporting the energy consumption, but it only uses one of the samples when reporting the power consumption. Unfortunately, the energy consumption is reported with too little accuracy (tenths of a kWh) to be useful in this work.

least 15 measurements of the total system power during kernel execution. Thus, we needed to ensure that the GPU was actively running kernels for at least 16 consecutive seconds. For the few applications we studied that have extremely long-running kernels, this was not a problem. However, most of the applications have kernels that run for less than one second, with some kernels running for less than a millisecond. For the applications that are inherently iterative, we set the number of iterations sufficiently high to keep the GPU occupied for the desired amount of time. For the other, non-iterative applications, we artificially forced a single kernel invocation to be repeated many times in rapid succession, with the number or repetitions dependent on the length of the kernel. We accomplished this using a program that interposed between the application and the OpenCL runtime. The program intercepted calls to the OpenCL function that launches a single kernel on the GPU (`clEnqueueNDRangeKernel`) and instead launched the kernel multiple times.

For each experiment, which consisted of running a single application with a fixed pair of core and memory frequencies, we first reset the power meter, then ran the application to completion, and finally downloaded the stored power measurements from the power meter. For most of our experiments, we were only interested in the average system power consumption during kernel execution, and so we needed to ignore power measurements that occurred either before or after the kernel executed. This process was greatly simplified by the fact that the maximum system power consumption occurred when the GPU was actively executing a kernel. Thus, to determine the average power consumption we first determined the number of seconds, N, that the GPU was actively executing and averaged the $N-2$ largest power values, after discarding the maximum value because it was extremely noisy. Manual inspection of power traces from a number of different applications confirmed the validity of this approach.

We determined the average execution time by dividing the total kernel execution time by the number of kernel invocations. For iterative applications, the total kernel execution time was reported by the application itself[11], while for the non-iterative applications the GPU

[11] For many applications, this time also included data transfer time. However, given the length of time that the kernels executed, the data transfer time was negligible for all applications.

execution time of each kernel invocation was reported by our interposing program. Once we obtained the average power consumption, P, and the average execution time, t, we computed the energy consumption as $E = Pt$.

5.2.2 Controlling for Temperature

Because the static (leakage) power of a processor is exponentially dependent on temperature, the power consumption of the GPU while executing a kernel does not solely depend on the properties of the kernel itself, but also on the initial temperature of the GPU. To ensure that our results were repeatable and represented a fair comparison of different frequencies, we ensured that the GPU temperature was the same before every experiment, where an experiment represented a single application running at fixed core and memory frequencies. For simplicity, we used as the default temperature the GPU's *idle temperature*, which we defined as the steady-state temperature of the GPU after it had remained idle for an indefinite period of time. The idle temperatures for the HD 5870 and HD 7970 were 38 °C and 42 °C, respectively. Throughout the experiments, the system was housed in a well-ventilated room in which the ambient temperature was maintained at a consistent 21 °C.

The idle temperature can be influenced by adjusting the GPU's fan speed, which is specified as a percentage of its maximum speed. By default the fan speed is increased or decreased automatically in response to changes in temperature. For both GPUs we used, the default fan speed when the GPU was at its idle temperature was 20%. Even after running under high load for a few minutes (during which the GPU temperature increased by as much 40 °C) we never observed a fan speed greater than 50%. For all of our experiments, we manually set the fan speed to 30%. To reduce the time spent between experiments waiting for the GPU to cool down, after each experiment ended we temporarily set the fan speed to 100% and then reset it to 30% once the GPU reached its idle temperature. For long-running applications, preliminary investigations suggest that manually setting the fan speed to 100% can yield non-trivial energy savings, because the decrease in leakage power

GPU	Domain	Frequencies (MHz)
HD 5870	Core	350 450 550 650 750 850
	Memory	200 300 500 600 700 800 900 1,000 1,100 1,200
HD 7970	Core	325 425 525 625 725 825 925
	Memory	175 275 375 475 575 675 775 875 975 1,075 1,175 1,275 1,375

Table 5.2: Core and memory clock frequencies used for frequency scaling.

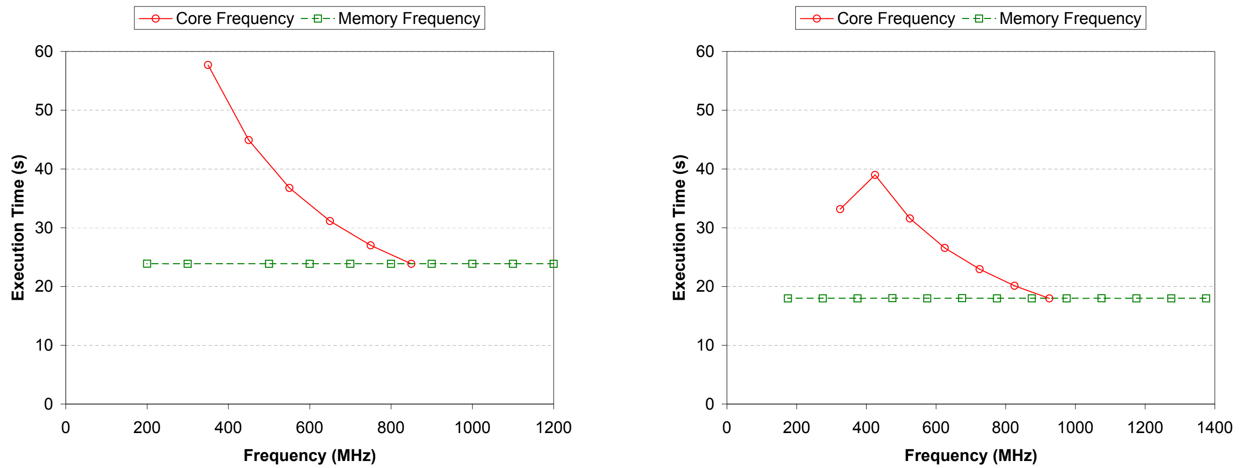
due to the decreased temperature outweighs the extra 10 to 15 W consumed by the fan itself. We leave further explorations of this tradeoff to future work.

5.3 Reducing GPU Energy Consumption: Potential

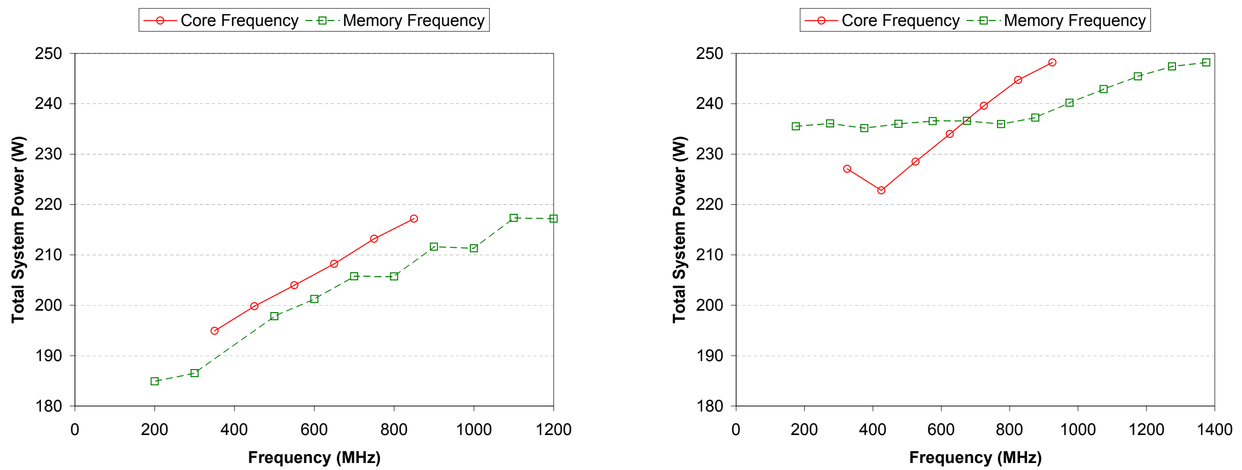
In this section, we demonstrate the potential energy savings of applying DVFS to GPUs by trying many pairs of core and memory frequencies and measuring the impact on energy consumption. For the purposes of this study, we only considered *decreasing* the frequencies from their default values, because increasing the frequencies can cause instability. Even so, the number of possible frequencies is enormous. On the HD 7970, for example, the core frequency can be set to any value in the range of 300 MHz to 925 MHz, while the memory frequency can be set to any value in the range of 150 MHz to 1375 MHz. If we were to consider every possible frequency, we would need to test 767,476 different frequency pairs. To make the problem tractable, we only decreased the frequencies from their default values in increments of 100 MHz. This left 61 and 91 frequency pairs for the HD 5870 and HD 7970, respectively. The frequencies we used are shown in Table 5.2.[12]

As a concrete example of the impact of changing frequencies, we will consider Bitonic Sort, a highly compute-bound application. Figure 5.1 shows Bitonic Sort's execution time, power consumption, and energy consumption on both GPUs as either the core or memory frequency was reduced. In later results, we explore all combinations of core and memory frequencies, but here we only vary one frequency at a time for simplicity. On both devices, reducing

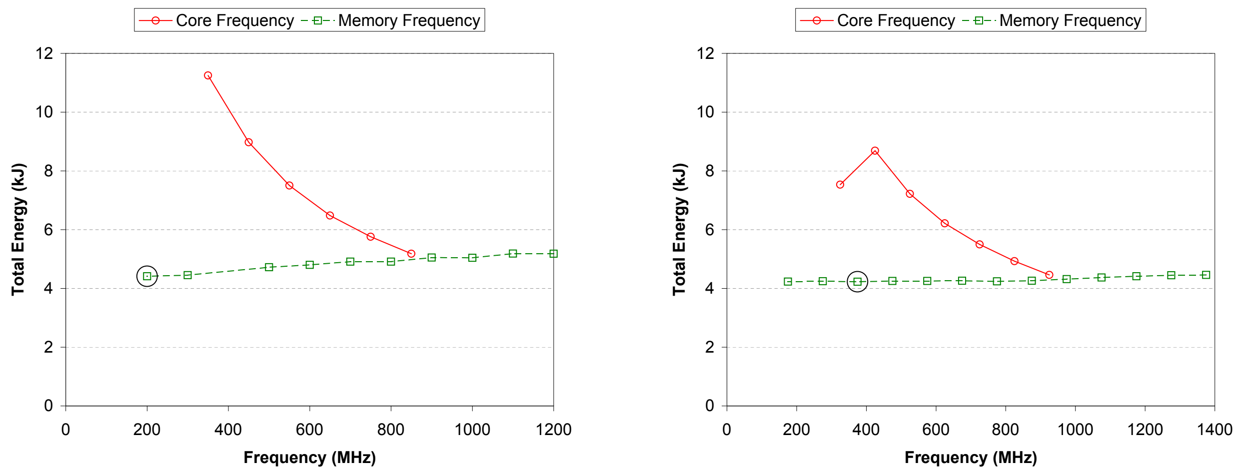
[12]For the HD 5870, a memory frequency of 400 MHz caused system instability and was not used.



(a) Execution time.



(b) Power consumption.



(c) Energy consumption.

Figure 5.1: Impact of varying the core and memory frequencies on the (a) execution time, (b) power consumption, and (c) energy consumption of Bitonic Sort running on the HD 5870 (left) and HD 7970 (right). The minimum energy configuration is circled.

the memory frequency had no impact on performance, while reducing the core frequency increased execution time by a little more than 2x in the worst case.[13] On the HD 5870, reducing either the core or memory frequency had a similar impact on power consumption, while on the HD 7970, the power consumption was significantly less sensitive to memory frequency, with consumption becoming effectively constant at memory frequencies below about half the default frequency. Because Bitonic Sort is compute-bound, this diminished sensitivity reduced the achievable energy savings to 5.2%, down from the 14.8% savings achievable with the HD 5870. For both GPUs, the energy savings from choosing the best frequency were much smaller than the energy *increase* from choosing the worst frequency: the worst frequencies for the HD 5870 and HD 7970 increased the energy consumption by 117% and 95%, respectively. Thus, it is entirely possible for a poorly constructed DVFS algorithm to increase rather than decrease the energy consumption of a system.

There are two strange phenomena present in the HD 7970 results in Figure 5.1: the power consumption's lack of sensitivity to changes in the memory frequency below 800 MHz, and the reversal of the execution time's upward trend (and the power consumption's downward trend) at the lowest core frequency. These data strongly suggest that, for low memory frequencies and the lowest core frequency, the GPU is not actually operating at the requested frequencies. Data from other applications, including those with memory-bound kernels, also supports this hypothesis. Unfortunately, there does not seem to be any way to directly confirm this assertion: when queried, the GPU reports that it is running at the requested frequencies.

To characterize the overall impact of changing frequencies, we measured the average execution time and power consumption of each application running at each frequency pair on both GPUs. Figure 5.2 shows the correlation between execution time and both core and memory frequency for the two GPUs. In both plots, points towards the upper-left corner represent compute-bound applications while points towards the lower-right corner represent memory-bound applications. Applications were well distributed between these two extremes

[13] Note that the two GPUs used different data sizes for Bitonic Sort, so the execution time and energy consumption cannot be directly compared.

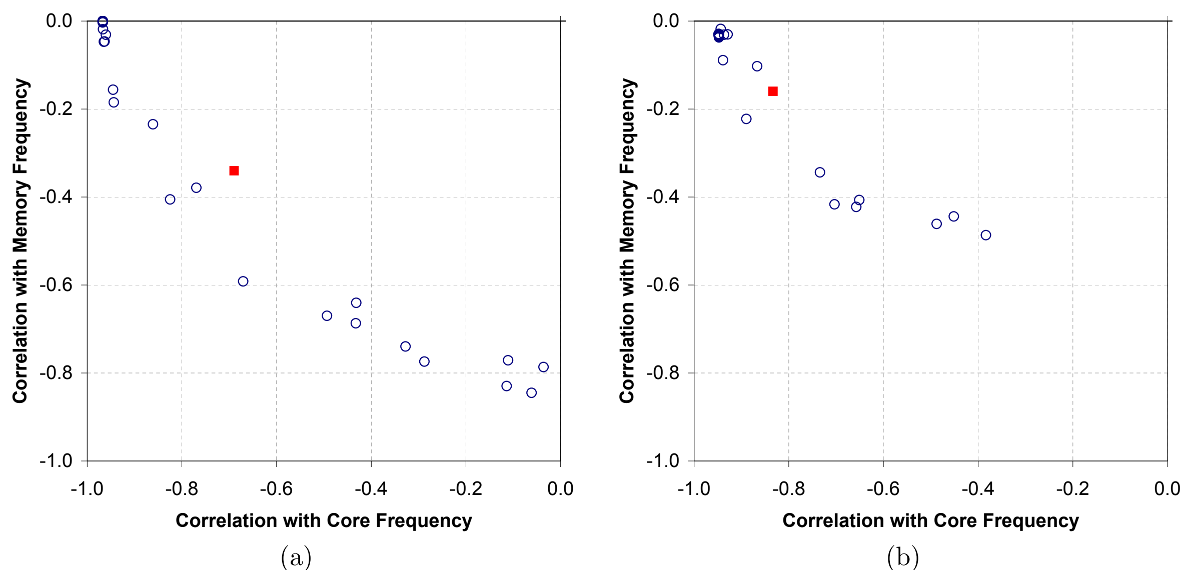


Figure 5.2: Pearson's correlation between execution time and the two frequencies for each application on the (a) HD 5870 and (b) HD 7970. The square represents the arithmetic mean across all applications. Note that all correlations are negative, because *decreasing* either the core or memory frequency logically only *increases* the execution time.

on the HD 5870, although on average they were around twice as sensitive to core frequency as they were to memory frequency. Overall, most applications were even more compute-bound on the HD 7970 and on average were four times as sensitive to core frequency as they were to memory frequency.

Figure 5.3 shows the correlation between power consumption and both core and memory frequency for the two GPUs. On the HD 5870, the power consumption of most applications was fairly sensitive to both frequencies and on average was slightly more sensitive to memory frequency. On the HD 7970, however, power consumption was on average about twice as sensitive to core frequency as it was to memory frequency.

Figure 5.4a shows the best observed reduction in compute energy for each application on the HD 5870, using three different approaches: allowing only the core frequency to vary, allowing only the memory frequency to vary, or allowing both frequencies to vary. The amount of energy savings varied significantly across different applications. For some applications, no energy savings were possible, while for other applications, the compute energy could be

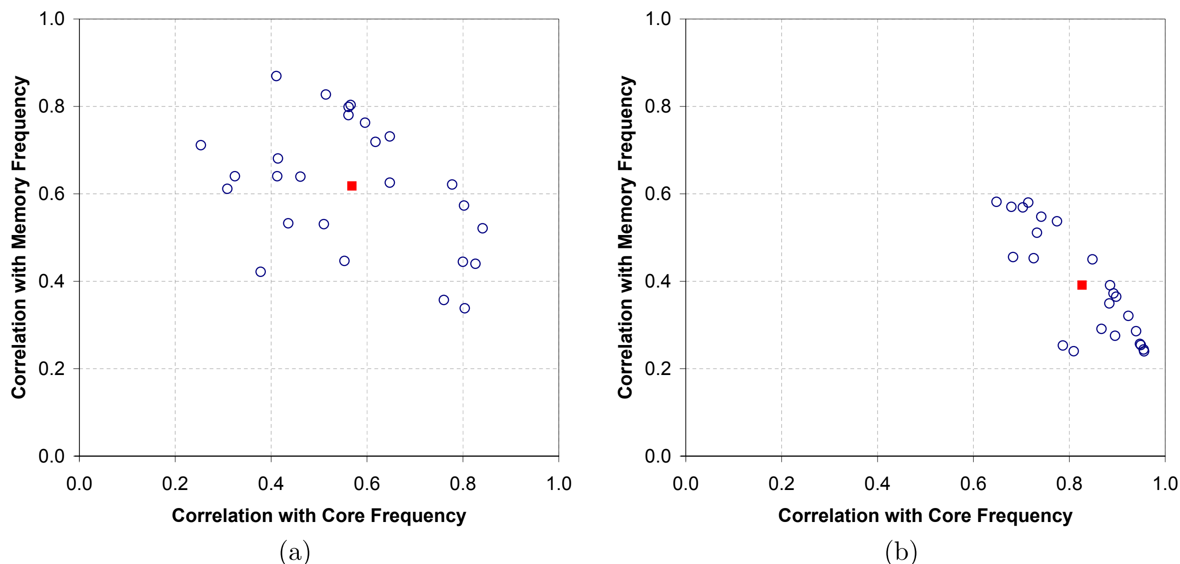
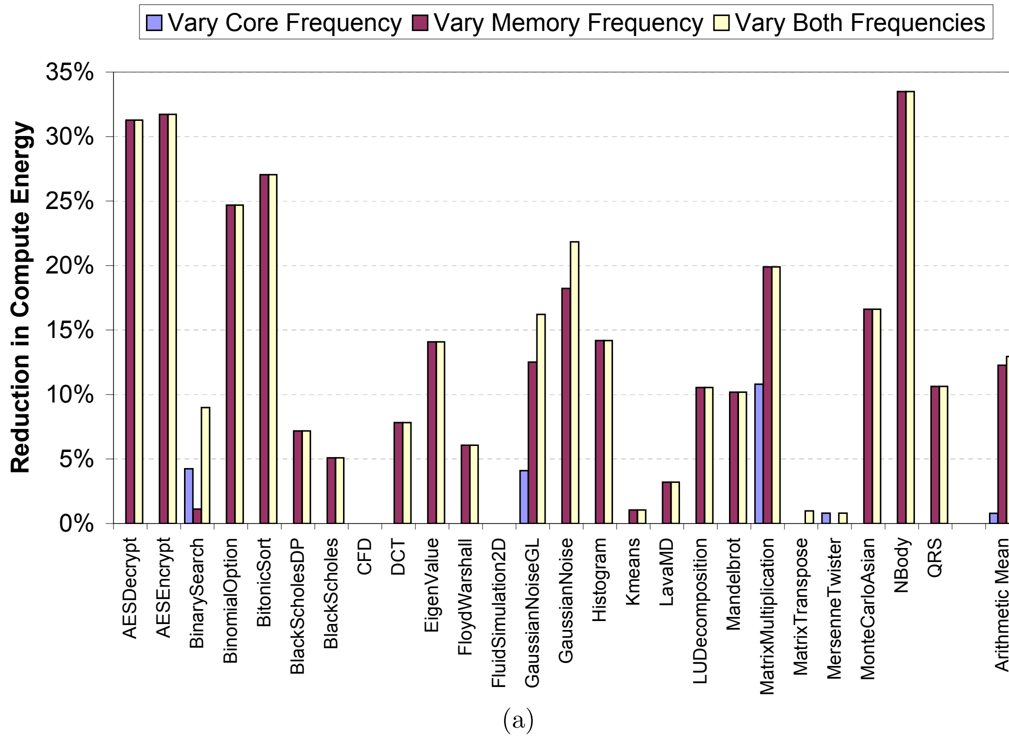


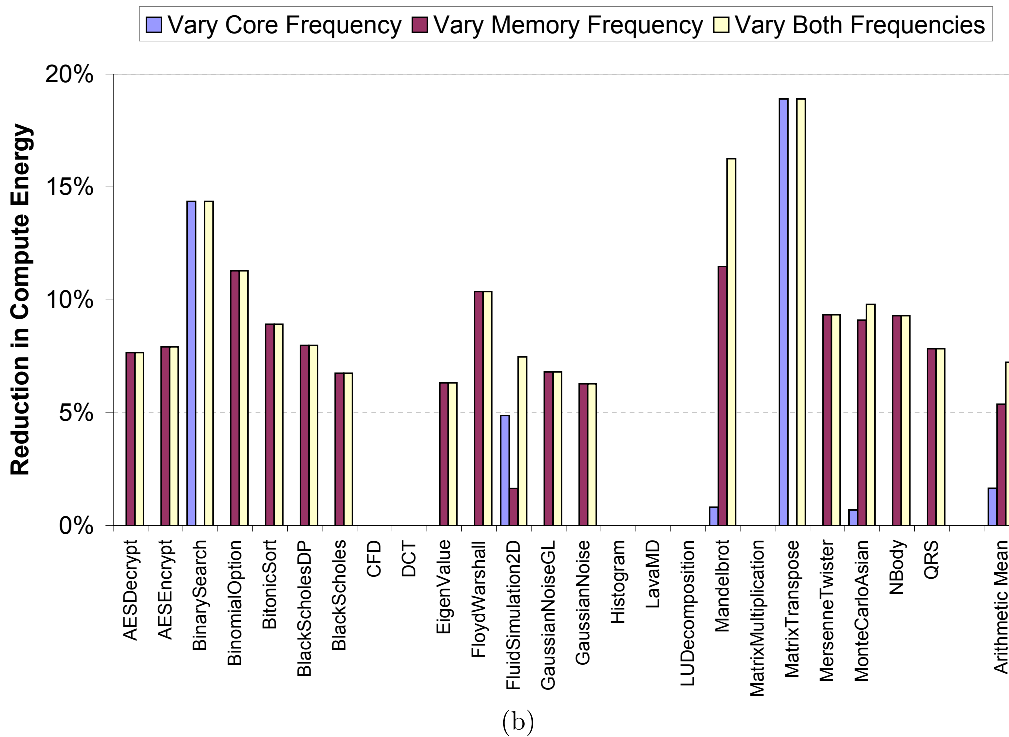
Figure 5.3: Pearson's correlation between total power consumption and the two frequencies for each application on the (a) HD 5870 and (b) HD 7970. The square represents the arithmetic mean across all applications. Note that all correlations are positive, because *decreasing* the core or memory frequency *decreases* the power consumption.

reduced by more than 30%. We saw earlier that, on average, the HD 5870's performance was most sensitive to core frequency while its power consumption was most sensitive to memory frequency. Thus, it should come as no surprise that varying the memory frequency provided much better energy savings than varying the core frequency. In fact, varying only the memory frequency was sufficient to obtain 95% of the optimal energy savings on average, while varying just the core frequency only provided 6% of the optimal savings. With the exception of Mersenne Twister, which achieved a negligible compute energy savings of 0.8%, no applications achieved their optimal savings by only varying the core frequency, while 18 applications achieved their optimal savings by only varying the memory frequency.

Figure 5.4b shows the optimal compute energy savings for the HD 7970. Although a few applications achieved greater energy savings than on the HD 5870, for most applications the achievable energy savings decreased. The number of applications which derived no benefit from DVFS increased from two to six, and overall the average energy savings decreased by 1.86x. We primarily attribute this change to the reduced sensitivity of power consumption to



(a)



(b)

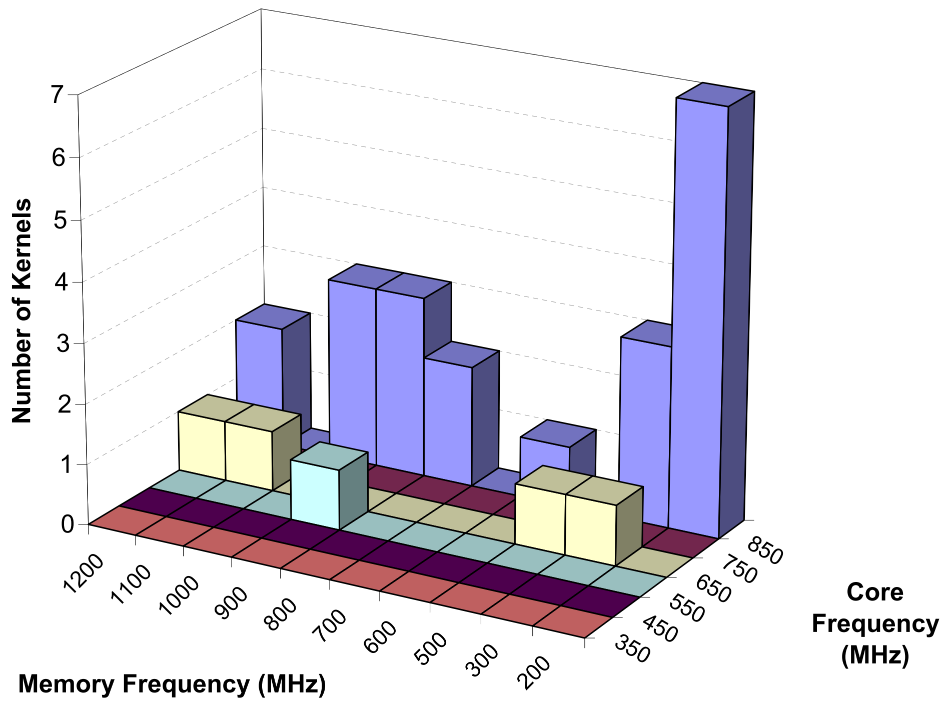
Figure 5.4: The optimal compute energy savings due to DVFS for the (a) HD 5870 and (b) HD 7970. The optimal savings are presented for three different approaches: only the core frequency is allowed to vary, only the memory frequency is allowed to vary, or both frequencies are allowed to vary. Note the different scales on the Y-axes.

the memory frequency that we described earlier. Despite this decreased sensitivity, adjusting memory frequency was still more useful than adjusting core frequency: varying only the memory frequency was sufficient to obtain 74% of the optimal energy savings, while varying only the core frequency provided 23% of the optimal savings. The increased benefits of adjusting the core frequency were primarily due to two extremely memory-bound applications, Binary Search and Matrix Transpose, which achieved larger energy savings than all but one other application. These two applications were clearly the exception: they were the only two that achieved their optimal energy savings by varying only the core frequency, while 13 applications achieved their optimal savings by only varying the memory frequency.

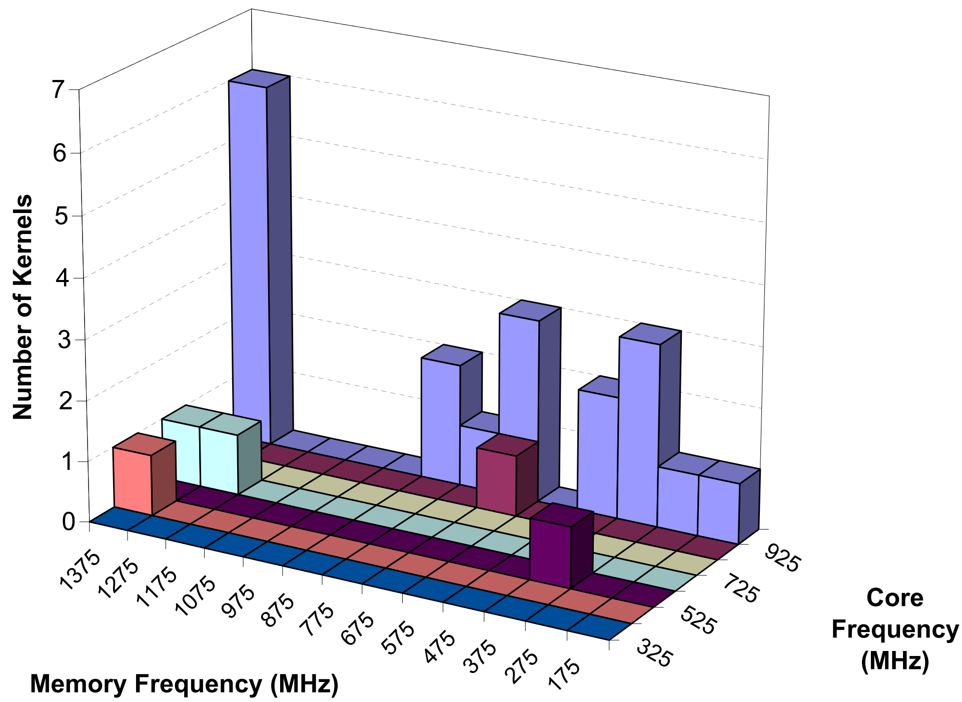
Figure 5.5 shows the optimal clock frequencies for all applications on both GPUs. These results are consistent with our earlier observation that reducing the memory frequency was more effective than reducing the core frequency for most applications. For the 5870, the pair of frequencies preferred by the largest number of applications was the maximum core frequency and the minimum memory frequency. For the 7970, on the other hand, the largest number of applications preferred the default frequencies (and therefore achieved no energy savings).

5.4 Reducing GPU Energy Consumption: Algorithms

The results presented so far have demonstrated the *potential* energy savings due to applying DVFS to GPUs. In this section, we explore techniques for applying DVFS in practice. In particular, the selection of the optimal frequency in the previous section relied on knowing the energy consumption of each configuration, which in turn required knowing the power consumption. In practice, it is unreasonable to assume that we will have access to accurate power consumption data, because this typically requires the existence of an external power meter. Even if a power meter exists, using it to get accurate power information can require changing an application's behavior in non-trivial ways (e.g., forcing a single kernel invocation



(a)



(b)

Figure 5.5: Optimal clock frequencies for each application on the (a) HD 5870 and (b) HD 7970.

to be repeated many times). Some newer NVIDIA GPUs, such as the Tesla K20, allow software to query the GPU's current power consumption. However, the measurements are only accurate to plus or minus five percent [76][14], which is significantly less accurate than measurements provided by stand-alone meters. Also, many of the implementation details, such as the sampling rate of the measurements and whether or not the power can be read concurrently with kernel execution, are left unspecified. Whether or not GPU-provided power measurements are sufficient for a DVFS algorithm is an open question, but one that will only be worth answering once hardware support is more widespread.

In this section, we first describe in detail the only existing DVFS algorithm for GPUs, GreenGPU. Then, to overcome the fundamental limitations of GreenGPU, we propose a new heuristic for selecting the most energy efficient clock frequencies, based on insights gleaned from the performance and energy data presented in the previous section.

5.4.1 GreenGPU

In prior work, Ma et al. [62] proposed GreenGPU, an online DVFS algorithm that attempts to gradually converge on the optimal core and memory frequencies based on the utilization levels reported by the GPU. The intuition behind the algorithm is that the frequency of a given component should scale with its utilization: as the utilization of the cores decreases, for example, reducing the core frequency should reduce power consumption without significantly impacting performance, thereby saving energy.

As mentioned earlier, the GreenGPU algorithm extends an existing CPU DVFS algorithm [31]. We first describe the original CPU version of the algorithm before explaining how the algorithm was extended in order to support GPUs. The CPU algorithm uses hardware performance counters to estimate the CPU utilization (or "CPU-intensiveness") of an application over a given interval, and assumes that the utilization level that can be supported is a linear function of the frequency. More specifically, the algorithm assumes that the maximum

[14] NVIDIA's documentation is inconsistent on the accuracy: another source claims an accuracy of plus or minus five Watts [75].

frequency can support a utilization of 100%, the minimum frequency can support a utilization of 0%[15], and the intermediate frequencies map linearly to intermediate utilization levels between 0% and 100%. Then, for each available frequency f, there are two cases. If the actual utilization is above the supported utilization of f, then setting the CPU's frequency to f would presumably increase execution time. If instead the actual utilization is below the supported utilization of f, then setting the CPU's frequency to f would presumably increase power consumption without improving performance, thereby increasing energy consumption. The theoretical performance loss and energy loss values (one if which must be zero) are combined into a single, overall loss value, based on a parameter, α, that determines the relative importance of performance and energy. The algorithm maintains a weight for each frequency, which is updated after each interval based on the overall loss value. In order to filter out the impact of transient events, the new weight is a function of both the old weight and the new loss value, and depends on a parameter, β, that controls the amount of hysteresis. The frequency with the best weight is chosen to be used in the next interval, and the process is repeated at the end of each interval.

GreenGPU extends this CPU algorithm to GPUs by computing the overall loss value for the cores and the memory separately[16] based on the core and memory utilizations, and then combining them together using a parameter, ϕ, that controls the relative importance of the cores and memory. All of the parameter values used in the GreenGPU results (α_C, α_M, β, and ϕ) were derived experimentally. The GreenGPU algorithm was developed and evaluated using an NVIDIA GeForce 8800 GTX GPU, which was released in 2006. The core and memory utilization values were measured using an NVIDIA tool, nvidia-smi, that reports the average utilizations at a one second granularity [75].[17] No equivalent tool is

[15] Mapping the minimum frequency to zero utilization seems like a strange choice; it might make more sense to map a frequency of zero to zero utilization. One result of this choice is that, on two platforms that are identical with the exception of the minimum allowable clock frequency, the algorithm may behave significantly differently.

[16] This requires separate values of α for the cores and memory, denoted α_C and α_M, respectively.

[17] In actuality, nvidia-smi does not report core utilization. What Ma et al. interpreted as the core utilization was actually the overall GPU utilization, which measures the fraction of time in the past second that the GPU was running a kernel. The two utilizations are not related in a meaningful way (except that core utilization

available for AMD GPUs, although similar information can be obtained for a single kernel invocation (rather than an arbitrary one-second interval) using AMD's APP Profiler, which reports various statistics based on hardware performance counters. Unfortunately, there is no mechanism to invoke the profiler at run time on a per-kernel basis. Instead, the entire application must be run through the profiler, and every kernel invocation is profiled, with the profiling data output to a file. To prevent this limitation from negatively impacting GreenGPU, we assumed that the profiling data can be acquired at run time with no overhead.

Although GreenGPU was proposed as an online algorithm that converges on a specific frequency pair over time, for the purposes of this study we were only interested in the frequency pair on which it eventually converged. To simulate the operation of the algorithm, we collected profiling data, using the AMD APP Profiler, for all applications and frequency pairs. At each iteration of the algorithm, GreenGPU requires two values, the core utilization and the memory utilization. For the core utilization, we used the *ALUBusy* counter on the HD 5870 and the *VALUBusy* counter on the HD 7970, which measure the percentage of time that a core[18] is processing non-memory (vector) instructions.[19] Measuring the memory utilization is more problematic, because AMD GPUs do not have any (publicly acknowledged) performance counters in the memory system. Our only alternative was to use performance counters in the core to approximate the memory system utilization: we used the *FetchUnitBusy* counter on the HD 5870 and the *MemUnitBusy* counter on the HD 7970, which measure the percentage of time that a core's fetch or memory unit is active, including stall time.

One might complain that the AMD performance counters are not a perfect match for the NVIDIA counters used in the original GreenGPU work. While such a complaint has merit,

---

can never exceed GPU utilization): running a compute-bound and a memory-bound kernel for the same amount of time would yield a vastly different core utilization but exactly the same GPU utilization. Because Ma et al. did not compare to the optimal savings, it is difficult to estimate how much this misunderstanding negatively impacted their results.

[18]Although the AMD APP Profiler documentation does not specify, we assume that the reported values are based on the performance counters of a single core (or compute unit, to use AMD's terminology).

[19]The ALUBusy counter on the HD 5870 never exceeded 50%, even for synthetic benchmarks. When simulating the GreenGPU algorithm, we scaled the value of this counter by a factor of two in order to make the peak utilization approximately 100%. This modification significantly improved GreenGPU's results.

it misses the bigger picture. What this discrepancy in available counters really highlights is the dangers of crafting an algorithm that relies too heavily on the features provided by a specific platform. The algorithm that we will propose in Section 5.4.3 only requires the execution time of the kernel, which, for OpenCL applications, is *guaranteed* to be provided by any OpenCL-compliant platform [49], and, for non-OpenCL applications, can easily be measured directly by the application itself.

5.4.2 Energy-Performance Tradeoff

Much of the prior work on CPU DVFS algorithms has claimed that there is a fundamental tradeoff between performance and energy, and that increasing the maximum tolerated loss in performance will typically increase the possible energy savings. For example, the four applications studied by Dhiman and Rosing [31] all had their largest energy savings with a performance loss of 25% or more, while the five applications studied by Choi et al. [27] all had their largest energy savings with a performance loss of around 30%. Both studies further claimed that the energy savings scaled roughly linearly with the tolerated performance loss. Dhiman and Rosing's results were based on the CPU DVFS algorithm [31] that was later extended by GreenGPU. The GreenGPU algorithm also assumes the validity of this energy-performance tradeoff[20] by incorporating a parameter (α) for explicitly controlling the relative importance of energy and performance.

Our findings indicate that high performance and low energy are inextricably linked for most GPU applications. Figure 5.6 plots, for each application, the increase in execution time versus the decrease in compute energy of the energy-optimal frequency pair. For the vast majority of applications, the optimal frequency pair increased execution time by less than five

[20] Ma et al. [62] actually contradict themselves on this point. On the one hand, they claim that "sometimes a DVFS setting with very low power consumption but a long execution time can be selected if its energy is the lowest" and that they therefore include the parameter α "to prevent this situation," specifically stating that "a *larger* α [emphasis added] directs the algorithm to favor energy saving." On the other hand, they also claim that "since energy increases when performance degrades ... we give a higher weight to performance by setting" α to an extremely *low* value. In other words, in order to optimize for energy, rather than using their algorithm's expressly-designed capability to prioritize energy consumption, they instead instruct their algorithm to prioritize performance over energy.

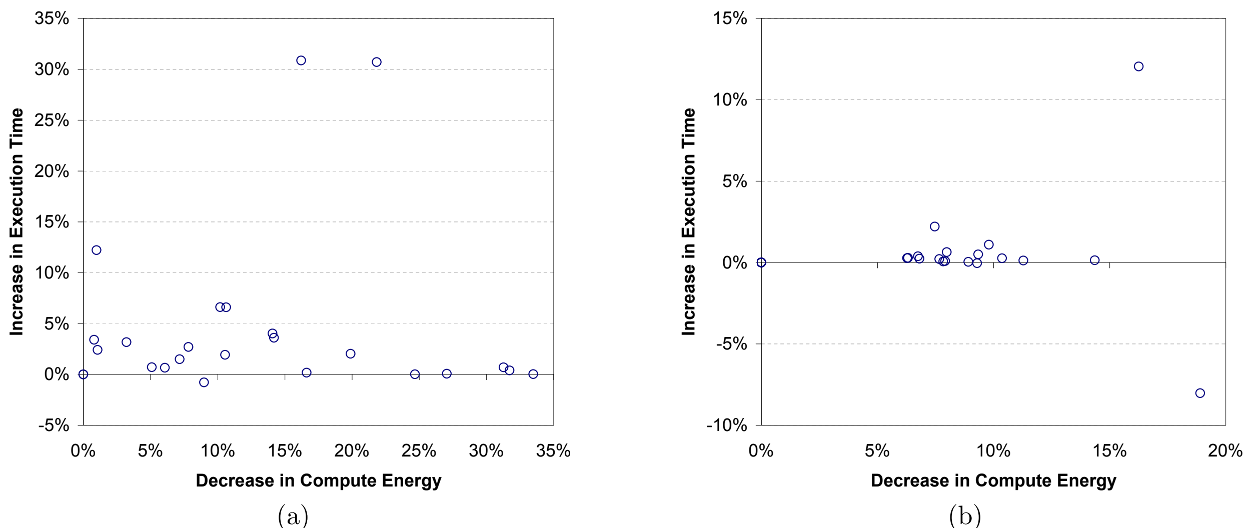


Figure 5.6: Increase in execution time versus decrease in compute energy for each optimal frequency pair on the (a) HD 5870 and (b) HD 7970.

percent. For the HD 5870, the arithmetic mean performance loss was 4.5%. Without the two outliers whose loss exceeded 30%, the average loss dropped to 2.3%. For the HD 7970, the arithmetic mean performance loss was only 0.4%. Excluding the applications that achieved no energy savings, and whose performance loss was by definition zero, the average loss was still only 0.6%.[21] For most applications, if our goal is to minimize energy consumption, we do not need to (and often cannot) tolerate large losses in performance.

5.4.3 ICE-AGE

We leverage this insight to design a simple but effective DVFS heuristic, which we call Ice-Age[22]. The goal of Ice-Age is to select the optimal frequency pair based only on the measured execution time for each pair. The two steps of the algorithm can be summarized succinctly as follows. First, throw out any frequency pair that results in an increase in

[21] The one outlier in the lower right corner of Figure 5.6b is Matrix Transpose, which is highly memory-bound but actually achieved slightly better performance when the core frequency was decreased.

[22] Ice-Age is an acronym which stands for "**I**ncreasing **C**ompute **E**fficiency **A**utomatically in **G**PU **E**nvironments."

execution time that is greater than some cutoff (e.g., 5%). Then, from the remaining frequency pairs, choose the one that has the lowest frequencies.

More formally, the algorithm is parameterized by a single value, C, which represents the performance cutoff. We define the execution time for a given core frequency f_C and memory frequency f_M to be $t\left(f_C, f_M\right)$. If the default frequencies are d_C and d_M, then we can compute the relative increase in execution time, Δt, using the following equation:

$$\Delta t\left(f_C, f_M\right) = \frac{t\left(f_C, f_M\right) - t\left(d_C, d_M\right)}{t\left(d_C, d_M\right)} = \frac{t\left(f_C, f_M\right)}{t\left(d_C, d_M\right)} - 1 \tag{5.1}$$

We throw out any pair $\left(f_C, f_M\right)$ with $\Delta t\left(f_C, f_M\right) > C$. In order to select a "minimum" pair of frequencies from the remaining pairs, we must guarantee that a minimum always exist by defining a total ordering on the set of frequency pairs. In most cases, the minimum of the remaining frequency pairs is obvious; for example, clearly $(550, 900) < (750, 1100)$ because $550 < 750$ and $900 < 1100$. Similarly, $(550, 900) < (550, 1100)$ because $550 \leq 550$ and $900 < 1100$. In rare cases, however, there is no obvious minimum frequency pair; for example, it is not immediately clear which of $(550, 900)$ and $(750, 600)$ should be considered smaller. To break ties in such cases, we choose the frequency pair with the smallest sum of frequencies; if that is still not sufficient, we choose the frequency pair with the minimum memory frequency. In practice, the specific method used to break ties has no impact on the final results, but we present it here for completeness.

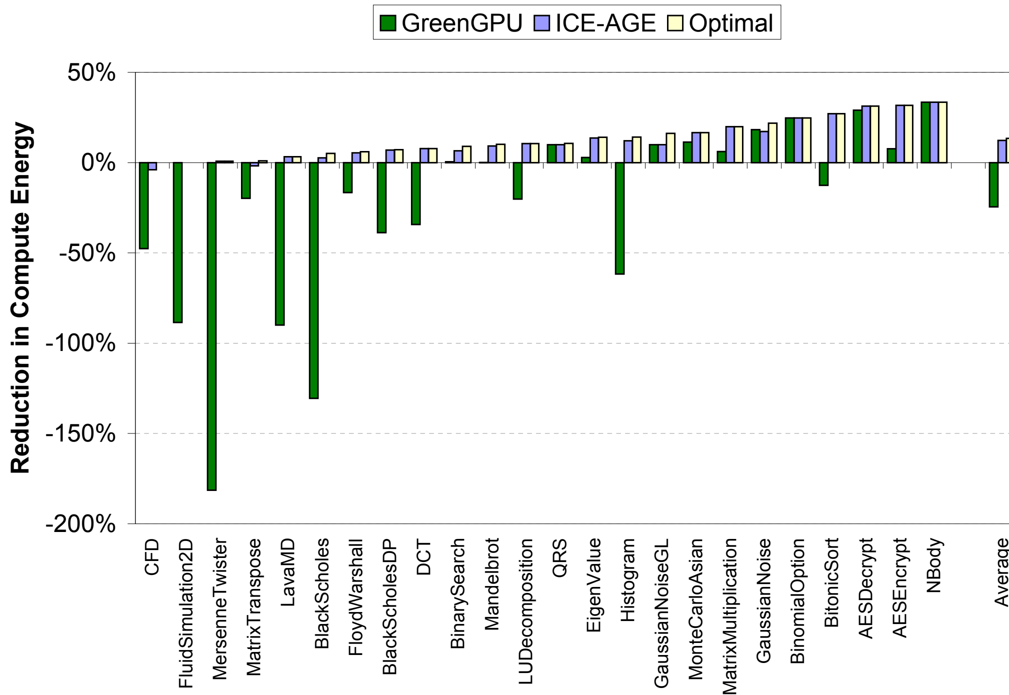
5.4.4 Results

We compared our proposed DVFS algorithm to both GreenGPU and the best possible energy savings achievable with complete access to power data. Although GreenGPU was designed to be used online, we assumed that both algorithms were used offline. Thus, we only cared about the frequency pair selected by each algorithm, and the energy savings achieved by an algorithm was simply the energy savings of its chosen frequency pair.

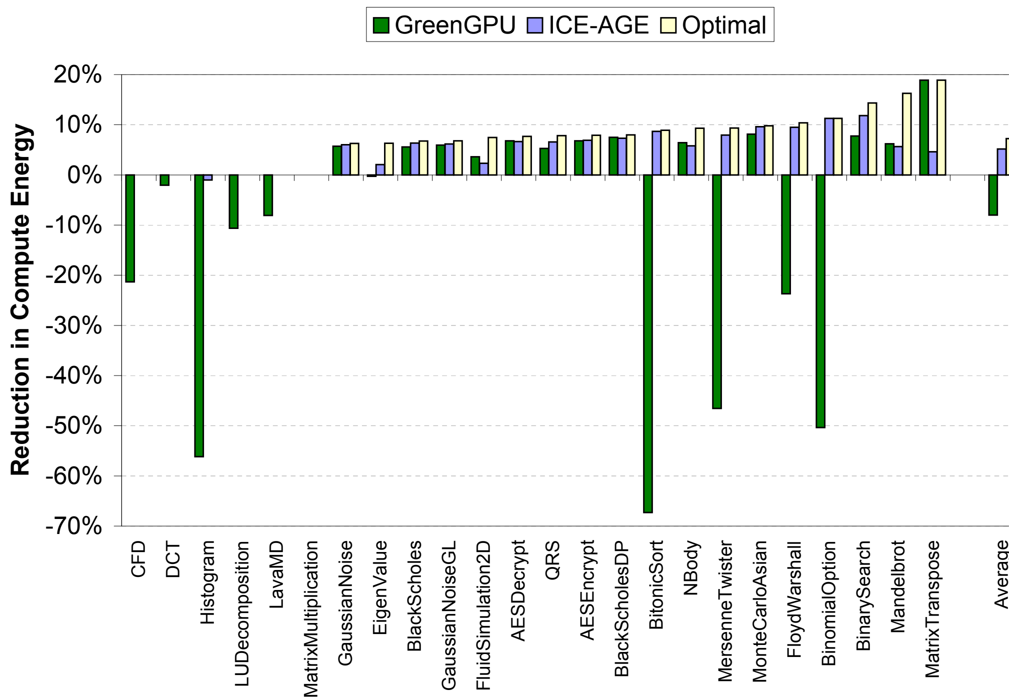
To derive the optimal values of GreenGPU's algorithmic parameters, we simulated the algorithm for all values of α_C, α_M, and ϕ between 0.00 and 1.00 (inclusive) in increments of 0.01, and chose the values that produced the best average energy savings across all applications. For both GPUs, the best values were $\alpha_C = 0.00$ and $\phi = 0.61$. The best values of α_M were 0.00 for the HD 5870 and 0.01 for the HD 7970. Assuming that the algorithm works as designed, these values make little sense because a value of $\alpha = 1$ optimizes for energy whereas a value of $\alpha = 0$ optimizes for performance. However, these values are consistent with the relatively low values used by Ma et al. [62] in their results ($\alpha_C = 0.15$ and $\alpha_M = 0.02$). Because we simulated the algorithm offline and were only interested in the frequency pair that the algorithm converged on, the precise value of β, which controls the hysteresis or rate of change of the weights, was essentially meaningless. We used the same value (0.2) as was used in the original GreenGPU results [62]. For each application, we simulated GreenGPU for 1,000 intervals, where each interval corresponded to a single kernel invocation. For most applications, the algorithm converged on a frequency pair within the first 10 intervals, but for a few applications it oscillated between two pairs.

The primary challenge in applying Ice-Age is selecting an appropriate value of C. Making the cutoff too low will exclude the most energy-efficient frequency pairs, while making the cutoff too high will include less energy-efficient pairs (which may be selected instead of the optimal pair if they use lower frequencies). To determine the optimal cutoff for a given GPU, we exhaustively tried a range of cutoffs from 0% to 15% in increments of 0.1% and chose the one that maximized the arithmetic mean energy savings across all applications. To reduce the chance of overfitting, we used leave-one-out cross validation. For each of the N applications, we computed its cutoff value C based on only the energy and performance data from the other $N - 1$ applications.

Figure 5.7a shows the compute energy savings achieved by GreenGPU and Ice-Age compared to the optimal energy savings on the HD 5870. Ice-Age provided better energy savings than GreenGPU for nineteen of the twenty-four applications, while GreenGPU only



(a)



(b)

Figure 5.7: Reduction in compute energy achieved by GreenGPU, Ice-Age, and the optimal for the (a) HD 5870 and (b) HD 7970, relative to running at the default frequencies. A negative value represents an *increase* in energy relative to the default frequencies.

bested Ice-Age for one application (and even then only by a negligible amount). Ice-Age matched the optimal savings on half of the applications, while GreenGPU matched the optimal on only two applications. The average compute energy savings for Ice-Age was 12.3%, while GreenGPU's average savings was -24.5%. In other words, GreenGPU consumed more energy than simply running the GPU at its default frequencies. While Ice-Age only increased energy consumption for two applications (and even then by only 3.9% in the worst case), GreenGPU increased the energy consumption for *half* of the applications. In the worst case, GreenGPU increased the energy consumption of Mersenne Twister by 181%.

Figure 5.7b shows the compute energy savings of the three approaches on the HD 7970. Ice-Age again significantly outperformed GreenGPU: Ice-Age provided better energy savings for seventeen of the applications, while GreenGPU only provided better energy savings for six applications (and only by a small amount for five of those six applications). GreenGPU's average savings was -8.0%, meaning that once again GreenGPU performed worse than simply running the GPU at its default frequencies. While Ice-Age increased energy consumption for only one application, GreenGPU increased energy for ten of the applications. On average, Ice-Age reduced the compute energy by 5.2%, versus an optimal savings of 7.2%. If we exclude the six applications for which even the optimal could not save energy, the average savings of Ice-Age and the optimal climbed to 7.0% and 9.6%, respectively. Ice-Age was primarily hurt by its performance on two applications, Mandelbrot and Matrix Transpose, which had the highest optimal energy savings but for which Ice-Age achieved more modest savings. The reduced savings for the latter two applications were due to a poor choice of the performance cutoff C. Those two applications were the only ones for which the leave-one-out cross validation selected a performance cutoff other than $C = 1.4\%$; using that cutoff for those two applications would have yielded much more substantial energy savings.

It is worth noting that, for both GPUs, GreenGPU's poor average energy savings were not purely due its large negative energy savings for some applications. Even if we artificially set all negative energy savings to zero, GreenGPU's average energy savings would still only

Figure 5.8: Ratio of actual energy savings to optimal energy savings as a function of the performance cutoff C.

be slightly less than half of the savings of Ice-Age for the HD 5870 and slightly more than half for the HD 7970.

As a measure of the sensitivity of Ice-Age to the performance cutoff C, Figure 5.8 shows the ratio of the average energy savings to the optimal energy savings for performance cutoffs ranging from 0% to 10%. A fairly large range of cutoffs yielded near-optimal savings for the HD 5870: any cutoff from 2.0% up to 4.9% allowed Ice-Age to achieve at least 90% of the optimal savings, and values as low as 1.0% allowed Ice-Age to reach at least 85% of the optimal. The maximum savings of 94% of the optimal occurred with cutoffs of 3.6% to 3.8%. The savings on the HD 7970 were much more sensitive to the performance cutoff, but there was still a substantial range of cutoffs that provided significant savings: Ice-Age achieved savings of at least 80% of the optimal for all cutoffs from 0.6% to 2.0%. The maximum savings of 83% of the optimal occurred at cutoffs of 1.4% to 2.0%. Although the shapes of the curves for the two GPUs are somewhat different, there is sufficient overlap between their

regions of highest savings to allow us to use a single cutoff value for both devices without sacrificing much of the achievable energy savings. Using a cutoff of 2.0% for both the HD 5870 and the HD 7970 yields energy savings of 90% and 83%, respectively. It is worth reiterating here that the performance cutoff represents the *maximum* tolerated performance loss; in practice, the actual performance loss was smaller. For example, using a cutoff of 2.0%, the average increase in execution time was only 0.89% for the HD 5870. For the HD 7970, the average increase in execution time was -0.02%, or 0.33% if we exclude Matrix Transpose.

5.5 Reducing CPU Energy Consumption

The GPU is the most obvious target for energy reduction techniques because it is the largest single power consumer in many systems. But during many phases of a GPU-enabled application, including GPU kernel execution, we can also apply frequency scaling to the CPU to obtain substantial energy savings. These energy savings are independent from and complementary to the GPU energy savings presented earlier. In fact, the potential energy savings for the CPU during kernel execution are often even larger than they are for the GPU. The energy savings achievable with GPU DVFS techniques are limited by the fact that reducing the GPU frequencies beyond a certain point has a direct impact on performance; in contrast, we can often drastically reduce the CPU's clock frequency without any significant performance consequences.

In most applications, when the GPU is actively executing a kernel, the CPU is simply waiting for the kernel to finish executing. Although waiting may not sound like a particularly power-hungry operation, it actually can be quite energy inefficient depending on whether it is implemented with blocking or non-blocking synchronization. Blocking synchronization means that the waiting CPU thread yields, allowing the CPU to either run other threads or idle, depending on whether other runnable threads are available. Then, at some point after the kernel completes, the CPU thread is awoken and can resume execution. Non-blocking

synchronization or busy waiting means that the CPU thread repeatedly checks if the kernel has completed. This typically lead to 100% CPU utilization, even though the CPU is not doing any useful work. Even worse, Linux's ondemand governor typically responds to this high utilization by increasing the CPU clock frequency to its maximum value. Whether blocking or busy waiting is better depends on the particular application as well as the metric being optimized. For short-running kernels, busy waiting can lead to significantly better performance, while for long-running kernels, the performance difference is minimal but blocking can significantly reduce power consumption.

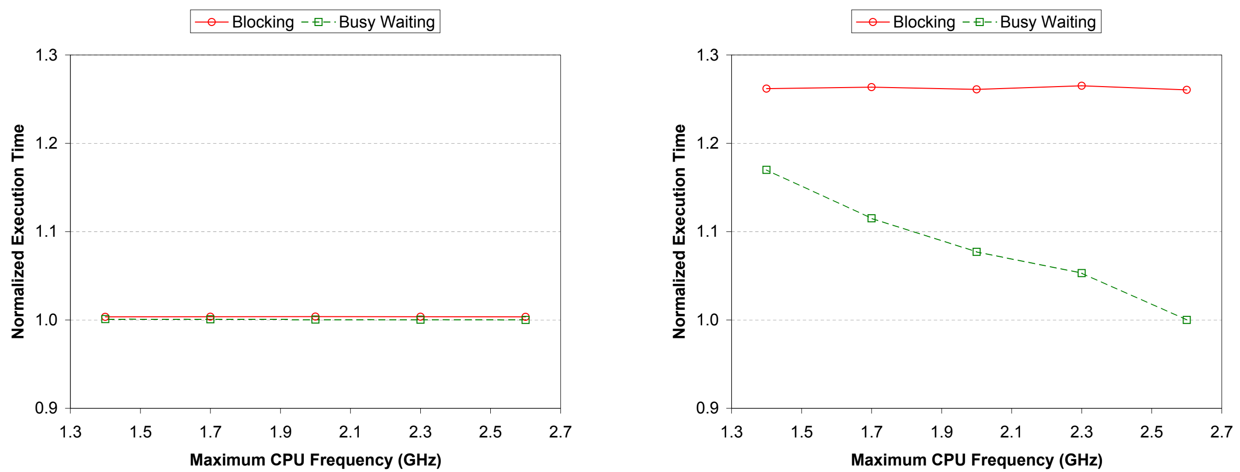
OpenCL provides two functions explicitly for the purpose of waiting for a kernel (or other operations) to complete: `clFinish`, which returns once all of the operations in a specified command queue have completed, and `clWaitForEvents`, which returns once a specified event or list of events has completed. In AMD's OpenCL implementation, both functions appear to use blocking synchronization[23] and appear to be equivalent in terms of both performance and power consumption. Instead of using the built-in OpenCL functions, most of the AMD APP SDK applications use busy waiting by repeatedly checking the status of the kernel until the kernel completes. This choice has significant power and energy implications.

For an application that uses busy waiting, we can use two approaches to reduce its power consumption. First, and most obviously, we can replace its busy waiting with blocking; here we use a call to `clFinish`. Second, we can reduce the maximum allowable CPU frequency[24] to limit the impact of high CPU utilization. Figure 5.9 shows the impact of these approaches on applications with vastly different kernel execution times, Bitonic Sort and CFD.

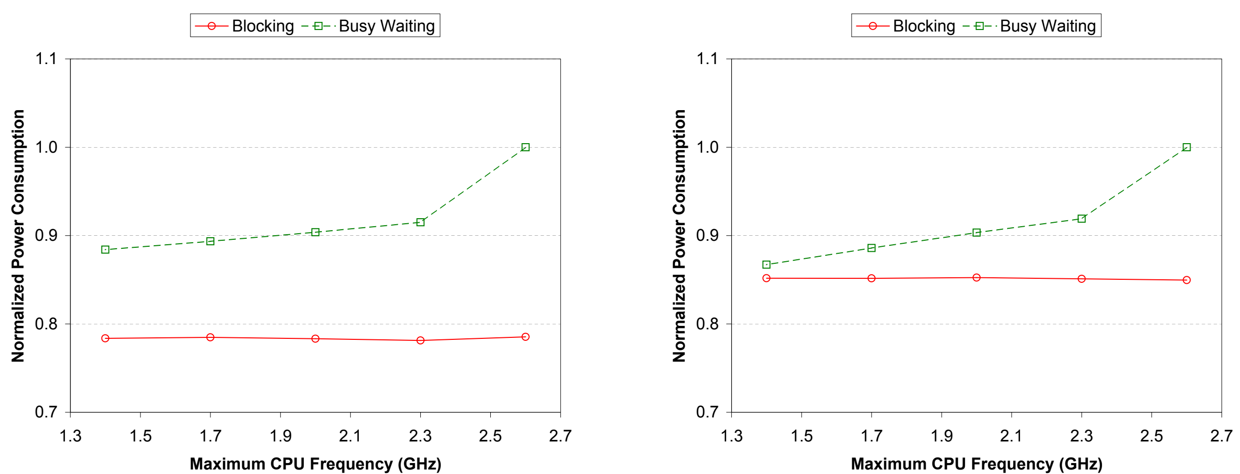
Each kernel invocation in Bitonic Sort executes for a relatively long time (approximately 80 milliseconds), which means that both approaches had a negligible impact on performance, increasing execution time by less than 0.4% in the worst case. At the same time, reducing the maximum CPU frequency from 2.6 GHz to 1.4 GHz reduced the average power consumption

[23] During a long-running kernel, the CPU utilization is essentially zero when using `clFinish` or `clWaitForEvents`.

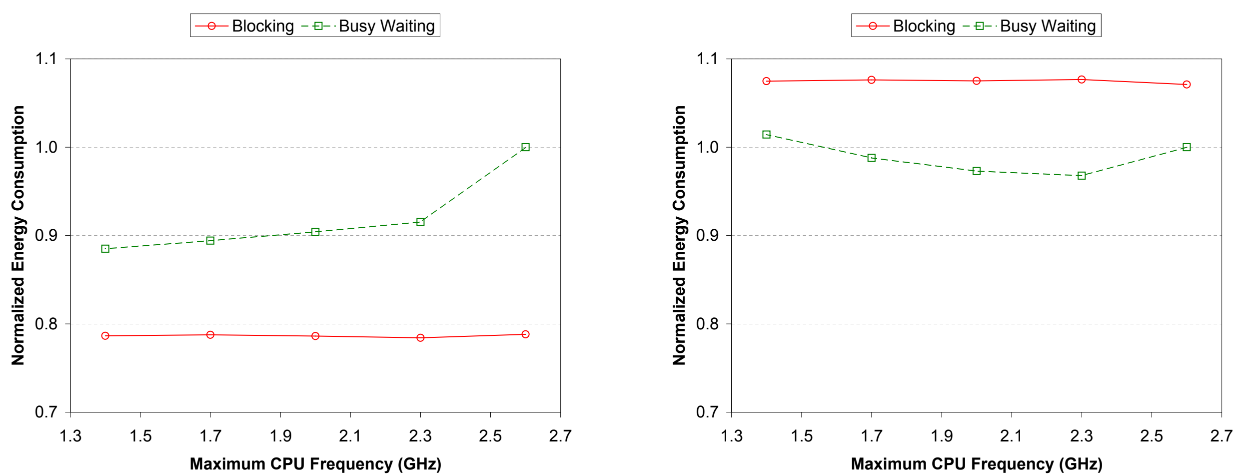
[24] This is easily achieved in Linux by simply writing the desired maximum frequency to a specific file.



(a) Execution time.



(b) Power consumption.



(c) Energy consumption.

Figure 5.9: Impact of different waiting methods and maximum CPU frequencies on the (a) execution time, (b) power consumption, and (c) energy consumption of Bitonic Sort (left) and CFD (right) running on the HD 5870. All metrics are normalized to busy waiting with a maximum CPU frequency of 2.6 GHz.

by 12%, while blocking instead of busy waiting reduced the power consumption by 22% irrespective of the maximum CPU frequency. Because of the negligible performance impact, the reduction in power of each approach led to an almost equivalent reduction in energy: minimizing the frequency reduced energy by 11% while using blocking reduced energy by 21%.

Unlike Bitonic Sort, CFD's kernels execute for an extremely short time (hundres of microseconds), and thus the impact of these techniques on performance was much greater. Reducing the maximum frequency to 1.4 GHz increased execution time by 17%, while switching to blocking increased execution time by 26% (once again, irrespective of frequency). Making matters worse, the reduction in power consumption due to switching to blocking was 15%, less than the reduction of 22% for Bitonic Sort. Overall, decreasing the maximum frequency to 1.4 GHz or switching to blocking both *increased* the total energy consumption, by 1.4% and 7.5%, respectively. However, reducing the maximum frequency by a smaller amount did result in energy savings: a maximum frequency of 2.3 GHz yielded a 3.2% reduction in energy consumption.

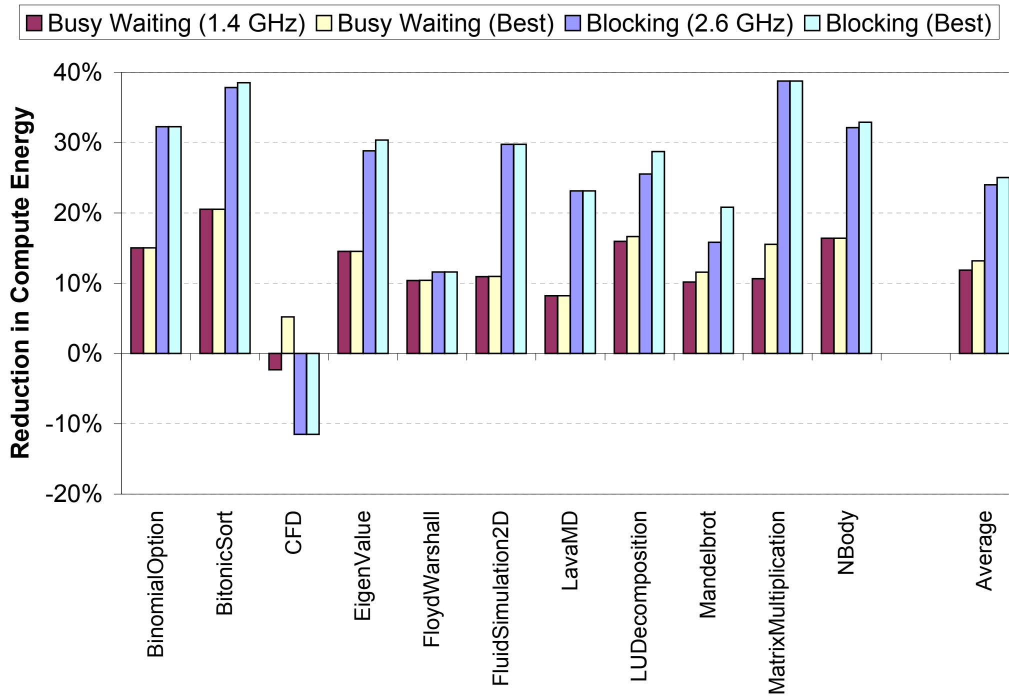
Due to the limitations of our power measurement infrastructure described in Section 5.2.1, the analysis shown above was not appropriate for all of the applications presented earlier. For example, a single kernel invocation in Black-Scholes executed for approximately 15 ms; to achieve an overall GPU execution time of approximately 15 seconds, we needed to invoke the kernel 1,000 times. Switching from waiting for the completion of one 15 milliseconds kernel to waiting for the completion of one *thousand* 15 milliseconds kernels is likely to have a non-negligible impact on the behavior of the CPU, especially with regards to automatic frequency scaling. Thus, for the experimental results presented in this section, we only considered applications for which long execution times represented a realistic use case. These applications fell into one of three categories: they were natively iterative (Bitonic Sort, CFD, Floyd-Warshall, and LU Decomposition), they naturally lent themselves to iterative behavior despite their lack of native support (Fluid Simulation 2D, LavaMD, Mandelbrot, and

N-Body[25]), or their kernel execution time was large enough that the small number of repeated kernel invocations had no meaningful impact on the CPU behavior (Binomial Option, Eigen Value, and Matrix Multiplication).

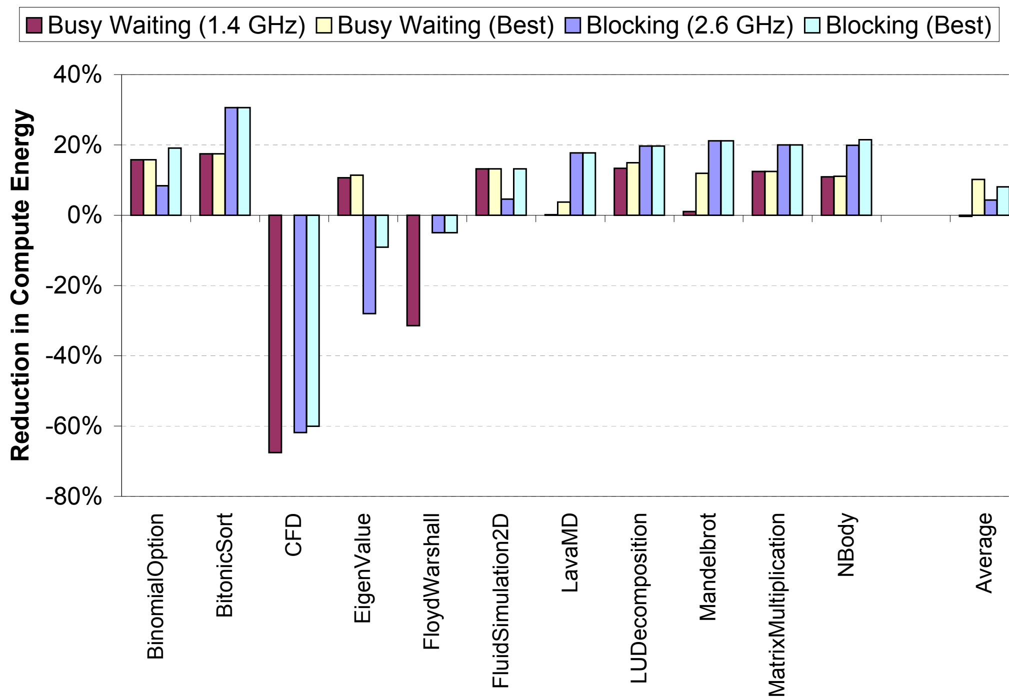
Figure 5.10a shows the compute energy savings achievable with the two CPU power reduction techniques, relative to busy waiting at the default maximum frequency, when using the HD 5870. For each application, we show the energy savings for four approaches: busy waiting with the maximum frequency set to its minimum value (1.4 GHz), blocking with the maximum frequency set to its default value (2.6 GHz), and both busy waiting and blocking at whichever maximum frequency yielded the greatest savings. For all applications except CFD, blocking provided better energy savings than simply reducing the maximum frequency. On average, blocking decreased compute energy by 22% while busy waiting with a reduced maximum frequency decreased compute energy by 11%. For some applications, using a maximum frequency less than 2.6 GHz improved the energy savings of blocking. Averaging across all of the applications, however, a maximum frequency of 2.6 GHz provided the best savings for blocking. Similarly, busy waiting with the maximum frequency set to 1.4 GHz provided better average savings than busy waiting at any other frequency, even though a different maximum frequency was more energy efficient for some applications.

Figure 5.10b shows the compute energy savings achievable when using the HD 7970. For almost all of the applications, the savings decreased relative to the HD 5870, for two reasons. First, switching to the HD 7970 reduced the kernel execution time for all applications, meaning that the negative performance impact of using these techniques increased. Second, the HD 7970 typically consumed more power than the HD 5870, so the relative power savings from using these techniques was reduced. Reducing the maximum frequency to 1.4 GHz while busy waiting significantly increased energy consumption for CFD and Floyd Warshall. Blocking at the default maximum frequency significantly increased energy consumption for

[25]Fluid Simulation 2D, Mandelbrot, and N-Body actually do support iteration natively, but they transfer data back to the CPU (unnecessarily) between each pair of successive kernel invocations. We avoided these extraneous data transfers by using our interposing program to achieve the iterative behavior, rather than using the native iterative support directly.



(a)



(b)

Figure 5.10: Compute energy savings for two waiting methods and different maximum CPU frequencies, relative to busy waiting with a maximum frequency of 2.6 GHz, when using the (a) HD 5870 or (b) HD 7970.

those two applications as well, and also slightly increased energy consumption for Eigen Value. For most applications, however, either reducing the maximum frequency while busy waiting or switching to blocking yielded significant energy savings, with blocking typically providing greater savings. Averaging across all of the applications, reducing the maximum frequency while busy waiting *increased* energy consumption by a negligible amount (0.4%) and blocking reduced energy consumption by 4.3%. In practice, of course, it would not make sense to use these techniques if they increase energy. Averaging across just the applications where these techniques were beneficial, reducing the maximum frequency and blocking reduced energy consumption by 11% and 18%, respectively. Interestingly, for both busy waiting and blocking, using a maximum frequency of 2.0 GHz provided the best average savings.

5.6 Related Work

The energy efficiency of GPUs has been studied from many different perspectives. A number of researchers have characterized the energy efficiency of specific applications running on GPU-enabled systems to determine whether it is more energy efficient to run the application on a CPU or a GPU [22, 41, 45, 46, 48, 53, 85, 86]. Takizawa et al. proposed automating this decision-making process with SPRAT, a method for automatically choosing the most energy efficient processor in a heterogeneous system at run time [99]. Other researchers have explored ways to improve the energy efficiency of GPU applications. Rather than considering the CPU and GPU as mutually exclusive choices, Ren and Suda manually divided a matrix multiply application across a CPU and multiple GPUs and characterized the impact on energy efficiency [83]. Ma et al. leveraged a previously developed statistical model to improve the energy efficiency of a kernel by automatically modifying its source code [64]. Zhang et al. measured the power consumption of different operations within a GPU application, including different types of data transfers, with the goal of providing application developers insight into the relative energy costs of different operations [108].

A number of statistical models have been developed for predicting a GPU's power consumption based on hardware performance counters [63, 68, 109]. With sufficient accuracy, we could leverage these models to improve our ability to predict which core and memory frequencies are the the most energy efficient. However, these models are targeted at hardware designers rather than application developers, and are too inaccurate to be of much practical benefit. More importantly, each model is highly tuned for a specific GPU and relies on performance counters that may not be available on other devices. We explicitly designed Ice-Age to be portable across multiple platforms.

Researchers have also developed more detailed GPU power models designed for use with architectural simulators [25, 42, 65, 93, 94]. They have used these models to estimate the impact of proposed GPU hardware changes on energy efficiency. We have instead focused our efforts on improving the energy efficiency of existing systems by leveraging capabilities (i.e., frequency scaling) that are already widely available.

5.7 Conclusions and Future Work

The use of DVFS to reduce CPU energy consumption has been an area of active research for nearly twenty years. Support for DVFS has become widely available in commodity hardware, and DVFS algorithms have sufficiently matured to be incorporated into major operating systems. The use of DVFS to reduce GPU energy, by contrast, is still a nascent area of research. The only previously proposed GPU DVFS algorithm relies on specific features provided by a particular vendor's GPUs; generalizing the algorithm to other platforms is both non-trivial and not effective.

We have proposed and evaluated a platform-independent DVFS algorithm for GPUs called Ice-Age, which is based on the observation that, for almost all applications, the best energy savings cannot be achieved without maintaining high performance. We have evaluated both the potential savings of DVFS and the actual savings achieved by Ice-Age using 24

diverse applications, the most powerful GPUs from two generations of AMD GPUs, and actual power measurements from a real system. We have shown that DVFS can reduce the compute energy for the two GPUs by up to 33% and 19%, respectively, in the best case and by 13% and 7%, respectively, on average. We have also shown that Ice-Age can achieve 94% and 83% of these optimal energy savings, respectively. Ice-Age is parameterized by only a single value, the performance cutoff or maximum tolerated performance loss. Although the optimal performance cutoff is slightly different for the two GPUs, using a performance cutoff of 2.0% for both devices is sufficient to achieve 87% of the optimal energy savings.

We have also demonstrated that the behavior of the CPU during GPU kernel execution can have a significant impact on the overall energy efficiency. Using a less power-hungry method of waiting for kernel completion (blocking rather than busy waiting) can reduce the compute energy for the two GPUs by 22% and 4%, respectively. Applying this technique more judiciously, only to applications where it provides a benefit, can increase the average compute energy savings to 25% and 13%, respectively. Automating the decision of when to use this technique and when not to use it is one obvious direction for future work. Another useful line of inquiry would be to study the impact of reducing the frequencies of both the GPU and CPU during data transfers between CPU and GPU memory. Preliminary investigations suggest that doing so can significantly reduce energy consumption.

For Ice-Age, we envision three primary directions for future work: measuring the effectiveness of Ice-Age across a wider range of GPUs, applying Ice-Age in an online rather than offline setting, and improving the ability of Ice-Age to predict the optimal frequency pair. We discuss of each of these directions in more detail below.

We claimed earlier that Ice-Age has been shown to work well on a wider range of GPUs than the only previously proposed GPU DVFS algorithm, GreenGPU [62]. This is certainly true, but is based on an extremely small sample size: GreenGPU has been evaluated on three GPUs, and shown to work well on only one, while Ice-Age has been evaluated on only two GPUs, but shown to work well on both. It would be worthwhile determining

whether or not Ice-Age also works well on a larger set of GPUs, including GPUs from a manufacturer other than AMD (i.e., NVIDIA). Unlike GreenGPU, Ice-Age does not rely on any platform-specific features in order to function; this is not to say, however, that is guaranteed to perform well on any platform. The fundamental question in evaluating the generality of Ice-Age is whether or not the link between high performance and low energy discussed in Section 5.4.2 holds true across a wide range of platforms.

We have evaluated Ice-Age in an offline context, in which we assumed that we had access to the execution time of the kernel of interest for all possible frequency pairs. In many cases it would be more useful to apply Ice-Age online to automatically adjust frequencies at run time. To make Ice-Age appropriate for online use, it would be necessary to decrease the number of frequency pairs that must be tested. This process is simplified by the observation that, in the vast majority of cases, execution time is a monotonically decreasing function of clock frequency. In other words, decreasing one of the clock frequencies almost never improves performance. Thus, if reducing the core frequency from 925 MHz to 825 MHz increases execution time by 5%, and our performance cutoff is 2%, there is no reason to consider decreasing the core frequency any further. For an application like Bitonic Sort, this insight alone reduces the number of frequency pairs that we would need to considered on the HD 7970 from 91 to 15.

Another promising approach for reducing the space of frequency pairs to be explored would be to use static analysis to determine whether a kernel is compute-bound or memory-bound. This information could be used to directly predict the optimal frequency pair or to narrow the number of frequency pairs to be tested. The determination of compute- or memory-boundedness could, for example, be based on a metric like the ratio of compute instructions to memory instructions. Of course, determining such a metric statically is not possible in the general case for an arbitrary kernel [84]. In practice, however, statically determining such a metric *is* possible for the vast majority of real-world kernels. As a proof-of-concept, rather than performing actual static analysis, we could leverage the dynamic profiling information

that we have already collected, using only the subset of statistics which conceivably could be collected statically. This would help us decide whether the approach holds promise and is worth pursuing further.

One of the goals of Ice-Age is to not rely on the features or capabilities specific to any particular platform. However, it is reasonable to assume that an algorithm that does leverage platform-specific information can outperform an algorithm that does not. It is likely that we could optimize the general, platform-independent form of the Ice-Age algorithm for a specific platform by augmenting it with platform-specific information. For example, when running on a system with an NVIDIA GPU, perhaps Ice-Age could leverage the memory utilization value used by GreenGPU to improve its frequency prediction.

Although in principle DVFS should be able to provide non-trivial energy savings for any GPU, in practice our ability to leverage DVFS for energy savings is dependent on vendor support. Vendors can choose to enable large energy savings by allowing scaling over a large frequency range, as AMD has done with the HD 5870. Vendors can also choose to limit the achievable energy savings by only allowing scaling over a narrower frequency range, as AMD appears to have done for the particular HD 7970 that we used. Finally, vendors can choose to completely prevent energy savings by disallowing frequency scaling altogether, as NVIDIA has done. By demonstrating the potential energy efficiency benefits of enabling DVFS, we hope to convince vendors to more fully support DVFS.

Chapter 6

Conclusions

Over the past decade, the semiconductor industry has come to rely heavily on improving performance through increased parallelism. Uniprocessors have given way to multi-core processors and modestly parallel GPUs have given way to massively parallel GPUs. As a result, leveraging concurrency has become critical for software to benefit from hardware improvements. Because graphics applications are inherently parallel, GPUs have benefited more than CPUs from the trend towards greater parallelism, at least as measured by peak throughput. As the gap in peak performance between CPUs and GPUs has grown, interest in leveraging GPUs for non-graphics applications has increased significantly. In response, manufacturers have made their GPUs increasingly general purpose and developed more programmer-friendly software interfaces. The performance, programmability, and ubiquity of GPUs has made them an attractive target for many high-performance applications.

Most applications that target GPUs do not efficiently use all of the available resources in a heterogeneous system. Applications might not utilize a resource at all; for example, in a system with two GPUs, a typical kernel would only execute on one of them. Or applications might utilize a resource, but not to its full potential; for example, a compute-bound kernel would underutilize the GPU's memory system. This underutilization hurts performance in the former case and hurts energy efficiency in the latter case. In this dissertation, we

have presented and evaluated techniques for addressing three different manifestations of this underutilization problem in the context of heterogeneous CPU-GPU systems.

In Chapter 3, we presented a comprehensive overview of the steps we took to accelerate a computationally demanding systems biology application, the detection and tracking of leukocytes. We observed that our initial implementations of the tracking algorithm were unable to keep the GPU fully utilized due to the overhead of launching so many separate kernels. Our solution was to abandon the canonical GPU parallelization approach in favor of a novel persistent kernel technique, thereby reducing the number of kernel calls dramatically and keeping the GPU fully utilized. We evaluated the application on a system with a high-end CPU and a high-end GPU, and showed that our GPU implementation was able to provide 26 times higher performance than our parallel CPU implementation. Based on our experiences, we presented a set of general guidelines for optimizing GPU applications. We also presented a set of recommendations for system-level changes that would simplify the development of high-performance GPU applications.

To improve the performance of GPU systems, in Chapter 4 we proposed an algorithm for automatically dividing the execution of an OpenCL kernel across multiple computational devices, so that existing and future OpenCL applications can better leverage all of the available devices in a heterogeneous system without significant programmer effort. Our proposed algorithm begins by scheduling a small subset of the available work on all devices to estimate their relative performance. It then schedules the remaining work based on this initial estimate, while taking into account any work that has already been scheduled but not yet completed.

We evaluated our algorithm using OpenCL applications from the AMD APP SDK and the Rodinia benchmark suite, running on a real hardware system with both an integrated and a discrete GPU. We compared the performance of our algorithm to that of the best possible fixed partition of work, which in practice would require extensive offline training to discover. Our scheduler nearly matched the average performance of this unrealistic upper

bound, without requiring any offline training. And when the performance of the underlying devices changed, our scheduler beat the best fixed partition by 10% on average and 20% in the worst case. Our scheduler can also provide resilience in the face of extreme performance imbalances caused by transient or permanent hardware or software failures.

To improve the energy efficiency of GPU systems, in Chapter 5 we explored the use of frequency scaling to slow down underutilized resources. We first showed that, with the ability to directly measure power consumption, non-trivial energy savings were achievable for two high-end AMD GPUs. However, it is unrealistic to assume access to power data in an arbitrary system. To overcome this limitation, we proposed Ice-Age, an algorithm for predicting the energy-optimal frequencies for an arbitrary kernel based solely on the execution times of the kernel at different frequencies. The design of Ice-Age was based on the observation that maintaining high performance is necessary for maximizing energy efficiency. Ice-Age first throws out any frequencies that result in a performance loss higher than some (small) cutoff, and then chooses the lowest frequency pair from the remaining frequencies. Using a single cutoff value across both GPUs, we showed that Ice-Age was able to achieve an average of 87% of the optimal compute energy savings.

We also showed that the CPU wastes considerable energy during GPU computation, by inefficiently busy waiting until the GPU completes execution. Reducing the CPU's clock frequency or forcing the CPU thread to block instead of busy wait can result in significant energy savings. However, these techniques can also increase energy consumption for applications with extremely short-lived kernels, so they must be applied with care.

As computer systems become more heterogeneous and diverse, keeping resources fully utilized becomes increasingly challenging. This places an ever greater burden on programmers, who must contend with differences among resources both within a single system and across multiple systems. As the trends towards greater heterogeneity continue for the foreseeable future, the types of automatic techniques for improving utilization that we presented in this dissertation will become increasingly crucial for software to realize the full benefits of future

hardware improvements. Thus, it is important that we work to both verify and improve the generality of these techniques across diverse software workloads and hardware systems.

Bibliography

[1] Alejandro Acosta, Robert Corujo, Vicente Blanco, and Francisco Almeida. Dynamic load balancing on heterogeneous multicore/multiGPU systems. In *International Conference on High Performance Computing and Simulation (HPCS)*, July 2010.

[2] ACPI. Advanced configuration and power interface specification (version 5.0). http://www.acpi.info/DOWNLOADS/ACPIspec50.pdf, 2011. Accessed on March 30, 2012.

[3] Scott T. Acton and Klaus Ley. Tracking leukocytes from in vivo video microscopy using morphological anisotropic diffusion. In *IEEE International Conference on Image Processing*, pages 300–303, 2001.

[4] Advanced Micro Devices. AMD accelerated parallel processing (APP) SDK. http://developer.amd.com/sdks/amdappsdk. Accessed on March 17, 2012.

[5] Advanced Micro Devices. AMD CrossFire. http://sites.amd.com/us/game/technology/Pages/crossfirex.aspx. Accessed on April 8, 2013.

[6] Advanced Micro Devices. AMD Radeon HD 7970 GHz edition. http://www.amd.com/US/PRODUCTS/DESKTOP/GRAPHICS/7000/7970GHZ/Pages/radeon-7970GHz.aspx. Accessed on April 7, 2013.

[7] Advanced Micro Devices. AMD launches world's fastest single-GPU graphics card – the AMD Radeon HD 7970. http://www.amd.com/us/press-releases/Pages/amd-launches-worlds-fastest-2011dec22.aspx, December 2011. Accessed on March 23, 2013.

[8] Advanced Micro Devices. AMD takes graphics crown with AMD Radeon HD 7970 GHz edition. http://www.amd.com/us/press-releases/Pages/amd-takes-graphics-2012jun22.aspx, June 2012. Accessed on April 7, 2013.

[9] Cédric Augonnet, Jérôme Clet-Ortega, Samuel Thibault, and Raymond Namyst. Data-aware task scheduling on multi-accelerator based platforms. In *International Conference on Parallel and Distributed Systems (ICPADS)*, December 2010.

[10] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40(12):33–37, December 2007.

[11] Anton Beloglazov, Rajkumar Buyya, Young Choon Lee, and Albert Zomaya. A taxonomy and survey of energy-efficient data centers and cloud computing systems. *Advances in Computers*, 82(2):47–111, 2011.

[12] Siegfried Benkner, Sabri Pllana, Jesper Larsson Traff, Philippas Tsigas, Uwe Dolinsky, Cédric Augonnet, Christoph Kessler, Beverly Bachmayer, David Moloney, and Vitaly Osipov. PEPPHER: Efficient and productive usage of hybrid computing systems. *IEEE MICRO*, 31(5):28–41, September 2011.

[13] Michael Boyer. Leukocyte detection & tracking: ImageJ plugin. http://www.cs.virginia.edu/~mwb7w/leukocyte/. Accessed on July 1, 2010.

[14] Michael Boyer, Shuai Che, Kevin Skadron, Jayanth Gummaraju, and Nuwan Jayasena. Automatic intra-application load balancing for heterogeneous systems. In *AMD Fusion Developer Summit*, June 2011.

[15] Michael Boyer, Shuai Che, Kevin Skadron, and Nuwan Jayasena. Load balancing in a changing world: Dealing with heterogeneity and performance variability. In *SRC TECHCON*, September 2012.

[16] Michael Boyer, Shuai Che, Kevin Skadron, Nuwan Jayasena, and Jayanth Gummaraju. Automatic intra-application load balancing for heterogeneous systems. In *SRC TECHCON*, September 2011.

[17] Michael Boyer and Kevin Skadron. Task sharing: Collaborative CPU-GPU execution. In *SRC TECHCON*, September 2010.

[18] Michael Boyer, Kevin Skadron, Shuai Che, and Nuwan Jayasena. Load balancing in a changing world: Dealing with heterogeneity and performance variability. In *Conference on Computing Frontiers*, May 2013. To appear.

[19] Michael Boyer, David Tarjan, Scott T. Acton, and Kevin Skadron. Accelerating leukocyte tracking using CUDA: A case study in leveraging manycore coprocessors. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, May 2009.

[20] David J. Brown and Charles Reams. Toward energy-efficient computing. *ACM Queue*, 8(2):30–43, February 2010.

[21] Juan M. Cebrían, Ginés D. Guerrero, and José M. García. Energy efficiency analysis of GPUs. In *Workshop on High-Performance, Power-Aware Computing (HPPAC)*, May 2012.

[22] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization (IISWC)*, October 2009.

[23] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing (JPDC)*, 68(10):1370–1380, October 2008.

[24] Shuai Che, Jeremy W. Sheaffer, Michael Boyer, Lukasz G. Szafaryn, Liang Wang, and Kevin Skadron. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In *IEEE International Symposium on Workload Characterization (IISWC)*, December 2010.

[25] Jianmin Chen, Bin Li, Ying Zhang, Lu Peng, and Jih-kwon Peir. Statistical GPU power analysis using tree-based methods. In *Workshop of Work in Progress in Green Computing*, July 2011.

[26] Long Chen, Oreste Villa, Sriram Krishnamoorthy, and Guang R. Gao. Dynamic load balancing on single- and multi-GPU systems. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, April 2010.

[27] Kihwan Choi, Ramakrishna Soma, and Massoud Pedram. Dynamic voltage and frequency scaling based on workload decomposition. In *International Symposium on Low Power Electronics and Design (ISLPED)*, 2004.

[28] Intel Corporation. Intel Xeon processor E7-8800/4800/2800 product families. http://www.intel.com/newsroom/kits/xeon/e7e3/gallery/gallery.htm. Accessed on March 23, 2013.

[29] Intel Corporation. Intel Xeon processor E7-8870. http://ark.intel.com/products/53580. Accessed on March 23, 2013.

[30] Mayank Daga, Ashwin M. Aji, and Wu-chun Feng. On the efficacy of a fused CPU+GPU processor (or APU) for parallel computing. In *Symposium on Application Accelerators in High-Performance Computing (SAAHPC)*, July 2011.

[31] Gaurav Dhiman and Tajana Simunic Rosing. Dynamic voltage frequency scaling for multi-tasking systems using online learning. In *International Symposium on Low Power Electronics and Design (ISLPED)*, 2007.

[32] Gregory Diamos and Sudhakar Yalamanchili. Harmony: An execution model and runtime for heterogeneous many core systems. In *International ACM Symposium on High Performance Distributed Computing (HPDC)*, June 2008.

[33] Gang Dong, Nilanjan Ray, and Scott T. Acton. Intravital leukocyte detection using the gradient inverse coefficient of variation. *IEEE Transactions on Medical Imaging*, 24(7):910–924, July 2005.

[34] Ecova Plug Load Solutions. 80 PLUS certified power supplies and manufacturers. http://www.plugloadsolutions.com/80PlusPowerSupplies.aspx, April 2013. Accessed on April 7, 2013.

[35] Zhe Fan, Feng Qiu, and Arie E. Kaufman. Zippy: A framework for computation and visualization on a GPU cluster. *Computer Graphics Forum*, 27(2):341–350, April 2008.

[36] S. B. Forlow, E. J. White, S. C. Barlow, S. H. Feldman, H. Lu, G. J. Bagby, A. L. Beaudet, D. C. Bullard, and K. Ley. Severe inflammatory defect and reduced viability in CD18 and E-selectin double-mutant mice. *Journal of Clinical Investigation*, 106(12):1457–1466, 2000.

[37] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel computing experiences with CUDA. *IEEE MICRO*, 28(4):13–27, July 2008.

[38] Chris Gregg, Michael Boyer, Kim Hazelwood, and Kevin Skadron. Dynamic heterogeneous scheduling decisions using historical runtime data. In *Workshop on Applications for Multi- and Many-Core Processors (A4MMC)*, June 2011.

[39] Chris Gregg, Jonathan Dorn, Kevin Skadron, and Kim Hazelwood. Fine-grained resource sharing for concurrent GPGPU kernels. In *USENIX Conference on Hot Topics in Parallelism (HotPar)*, June 2012.

[40] Jayanth Gummaraju, Laurent Morichetti, Michael Houston, Ben Sander, Benedict R. Gaster, and Bixia Zheng. Twin Peaks: A software platform for heterogeneous computing on general-purpose and graphics processors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2010.

[41] Dominik Gddeke, Robert Strzodka, Jamaludin Mohd-Yusof, Patrick McCormick, Sven H.M. Buijssen, Matthias Grajewski, and Stefan Turek. Exploring weak scalability for FEM calculations on a GPU-enhanced cluster. *Parallel Computing*, 33(10-11):685–699, November 2007.

[42] Sunpyo Hong and Hyesoon Kim. An integrated GPU power and performance model. In *International Symposium on Computer Architecture (ISCA)*, June 2010.

[43] Yoshihiko Hotta, Mitsuhisa Sato, Hideaki Kimura, Satoshi Matsuoka, Taisuke Boku, and Daisuke Takahashi. Profile-based optimization of power performance by using dynamic voltage scaling on a PC cluster. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, April 2006.

[44] Qiming Hou, Kun Zhou, and Baining Guo. SPAP: A programming language for heterogeneous many-core systems. Technical report, Zhejiang University Graphics and Parallel Systems Lab, January 2010.

[45] Song Huang, Shucai Xiao, and Wu-chun Feng. On the energy efficiency of graphics processing units for scientific computing. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, May 2009.

[46] Yang Jiao, Heshan Lin, Pavan Balaji, and Wu-chun Feng. Power and performance characterization of computational kernels on the GPU. In *IEEE/ACM International Conference on Green Computing and Communications (GreenCom)*, December 2010.

[47] Ali Keshavarzi, Kaushik Roy, and Charles F. Hawkins. Intrinsic leakage in low power deep submicron CMOS ICs. In *International Test Conference*, November 1997.

[48] Srinidhi Kestur, John D. Davis, and Oliver Williams. BLAS comparison on FPGA, CPU and GPU. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, July 2010.

[49] Khronos OpenCL Working Group. The OpenCL specification (version 1.2, document revision 19). http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf, November 2012. Accessed on March 31, 2013.

[50] Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. Achieving a single compute device image in OpenCL for multiple GPUs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, February 2011.

[51] Clyde P. Kruskal and Alan Weiss. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software Engineering*, 11(10):1001–1016, October 1985.

[52] George Kyriazis. Heterogeneous system architecture: A technical review. http://developer.amd.com/wordpress/media/2012/10/hsa10.pdf, August 2010.

[53] Holger Lange, Florian Stock, Andreas Koch, and Dietmar Hildenbrand. Acceleration and energy efficiency of a geometric algebra computation using reconfigurable computers and GPUs. In *IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*, April 2009.

[54] LAVA Lab at the University of Virginia. The Rodinia benchmark suite. http://lava.cs.virginia.edu/wiki/rodinia. Accessed on March 17, 2012.

[55] Jungseob Lee, Vijay Sathisha, Michael Schulte, Katherine Compton, and Nam Sung Kim. Improving throughput of power-constrained GPUs using dynamic voltage/frequency and core scaling. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 2011.

[56] Jian Li, José F. Martinez, and Michael C. Huang. The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors. In *International Symposium on High-Performance Computer Architecture (HPCA)*, February 2004.

[57] Min Yeol Lim, Vincent W. Freeh, and David K. Lowenthal. Adaptive, transparent frequency and voltage scaling of communication phases in MPI programs. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, November 2006.

[58] Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. Meng. Merge: A programming model for heterogeneous multi-core systems. *SIGOPS Operating Systems Review*, 42(2):287–296, March 2008.

[59] Cong Liu, Jian Li, Wei Huang, Juan Rubio, Evan Speight, and Xiaozhu Lin. Power-efficient time-sensitive mapping in heterogeneous systems. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2012.

[60] Los Alamos National Laboratory. End of the road for Roadrunner. http://www.lanl.gov/newsroom/news-releases/2013/March/03.29-end-of-roadrunner.php, March 2013. Accessed on April 8, 2013.

[61] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, December 2009.

[62] Kai Ma, Xue Li, Wei Chen, Chi Zhang, and Xiaorui Wang. GreenGPU: A holistic approach to energy efficiency in GPU-CPU heterogeneous architectures. In *International Conference on Parallel Processing (ICPP)*, September 2012.

[63] Xiaohan Ma, Mian Dong, Lin Zhong, and Zhigang Deng. Statistical power consumption analysis and modeling for GPU-based computing. In *ACM SOSP Workshop on Power Aware Computing and Systems (HotPower)*, October 2009.

[64] Xiaohan Ma, Marion Rincon, and Zhigang Deng. Improving energy efficiency of GPU based general-purpose scientific computing through automated selection of near optimal configurations. Technical Report UH-CS-11-08, University of Houston Department of Computer Science, October 2011.

[65] Bren Mochocki, Kanishka Lahiri, and Srihari Cadambi. Power analysis of mobile 3D graphics. In *Conference on Design, Automation and Test in Europe (DATE)*, March 2006.

[66] Adam Moerschell and John D. Owens. Distributed texture memory in a multi-GPU environment. In *Graphics Hardware 2006*, September 2006.

[67] Christoph Muller, Steffen Frey, Magnus Strengert, Carsten Dachsbacher, and Thomas Ertl. A compute unified system architecture for graphics clusters incorporating data locality. *IEEE Transactions on Visualization and Computer Graphics*, 15(4):605–617, July/August 2009.

[68] Hitoshi Nagasaka, Naoya Maruyama, Akira Nukada, Toshio Endo, and Satoshi Matsuoka. Statistical power modeling of GPU kernels using performance counters. In *International Green Computing Conference (IGCC)*, August 2010.

[69] Umesh Gajanan Nawathe, Mahmudul Hassan, King C. Yen, Ashok Kumar, Aparna Ramachandran, and David Greenhill. Implementation of an 8-core, 64-thread, power-efficient SPARC server on a chip. *Journal of Solid State Circuits*, 43(1):6–20, January 2008.

[70] Andrew Nere, Atif Hashmi, and Mikko Lipasti. Profiling heterogeneous multi-GPU systems to accelerate cortically inspired learning algorithms. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, May 2011.

[71] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, March 2008.

[72] NVIDIA. SLI technology. http://www.geforce.com/hardware/technology/sli/technology. Accessed on April 8, 2013.

[73] NVIDIA. NVIDIA CUDA programming guide (version 2.2.1), May 2009.

[74] NVIDIA. NVIDIA CUDA C best practices guide (version 5.0). http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html, October 2012. Accessed on April 7, 2013.

[75] NVIDIA. nvidia-smi – NVIDIA system management interface program (version 4.304). https://developer.nvidia.com/sites/default/files/akamai/cuda/files/CUDADownloads/NVML_cuda5/nvidia-smi.4.304.pdf, October 2012. Accessed on March 28, 2013.

[76] NVIDIA. NVML API reference manual (version 4.304.55). https://developer.nvidia.com/sites/default/files/akamai/cuda/files/CUDADownloads/NVML_cuda5/nvidia-smi.4.304.pdf, October 2012. Accessed on March 31, 2013.

[77] OpenMP Architecture Review Board. OpenMP application program interface (version 2.5). http://www.openmp.org/mp-documents/spec25.pdf, May 2005. Accessed on March 19, 2013.

[78] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Tim Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, March 2007.

[79] Venkatesh Pallipadi and Alexey Starikovskiy. The ondemand governor: Past, present, and future. In *Linux Symposium*, July 2006.

[80] Constantine D. Polychronopoulos and David J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, 36(12):1425–1439, December 1987.

[81] Nilanjan Ray and Scott T. Acton. Motion gradient vector flow: An external force for tracking rolling leukocytes with shape and size constrained active contours. *IEEE Transactions on Medical Imaging*, 23(12):1466–1478, December 2004.

[82] Nilanjan Ray, Scott T. Acton, and Klaus Ley. Tracking leukocytes in vivo with shape and size constrained active contours. *IEEE Transactions on Medical Imaging*, 21(10):1222–1235, October 2002.

[83] Da Qi Ren and Reiji Suda. Investigation on the power efficiency of multi-core and GPU processing element in large scale SIMD computation with CUDA. In *International Green Computing Conference (IGCC)*, August 2010.

[84] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, March 1953.

[85] Mahsan Rofouei, Thanos Stathopoulos, Sebi Ryffel, William Kaiser, and Majid Sarrafzadeh. Energy-aware high performance computing with graphic processing units. In *Conference on Power Aware Computing and Systems (HotPower)*, December 2008.

[86] David Rohr, Matthias Bach, Matthias Kretz, and Volker Lindenstruth. Multi-GPU DGEMM and high performance Linpack on highly energy-efficient clusters. *IEEE Micro*, 31(5):18–27, September 2011.

[87] Philip E. Ross. A computer for the clouds. *IEEE Spectrum*, August 2008.

[88] Barry Rountree, David K. Lowenthal, Shelby Funk, Vincent W. Freeh, Bronis R. de Supinski, and Martin Schulz. Bounding energy consumption in large-scale MPI programs. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, November 2007.

[89] Barry Rountree, David K. Lownenthal, Bronis R. de Supinski, Martin Schulz, Vincent W. Freeh, and Tyler Bletsch. Adagio: Making DVS practical for complex HPC applications. In *International Conference on Supercomputing (ICS)*, June 2009.

[90] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-Zee Ueng, John A. Stratton, and Wen-mei W. Hwu. Program optimization space pruning for a multithreaded GPU. In *International Symposium on Code Generation and Optimization (CGO)*, April 2008.

[91] Mark Segal and Mark Peercy. A performance-oriented data parallel virtual machine for GPUs. In *International Conference and Exhibition on Computer Graphics and Interactive Techniques (SIGGRAPH)*, July 2006.

[92] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A many-core x86 architecture for visual computing. *ACM Transactions on Graphics*, 27(3):1–15, 2008.

[93] Jeremy W. Sheaffer, David Luebke, and Kevin Skadron. A flexible simulation framework for graphics architectures. In *ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, August 2004.

[94] Jeremy W. Sheaffer, Kevin Skadron, and David Luebke. Studying thermal management for graphics-processor architectures. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2005.

[95] Chun-Yu Shei, Pushkar Ratnalikar, and Arun Chauhan. Automating GPU computing in MATLAB. In *International Conference on Supercomputing (ICS)*, May 2011.

[96] Jie Shen, Jianbin Fang, Henk Sips, and Ana Lucia Varbanescu. Performance traps in OpenCL for CPUs. In *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, February 2013.

[97] Reiji Suda and Da Qi Ren. Accurate measurements and precise modeling of power dissipation of CUDA kernels toward power optimized high performance CPU-GPU computing. In *International Conference on Parallel and Distributed Computing, Applications and Technologies*, December 2009.

[98] Enqiang Sun, Dana Schaa, Richard Bagley, Norman Rubin, and David Kaeli. Enabling task-level scheduling on heterogeneous platforms. In *Workshop on General Purpose Processing with Graphics Processing Units (GPGPU)*, March 2012.

[99] Hiroyuki Takizawa, Katuto Sato, and Hiroaki Kobayashi. SPRAT: Runtime processor selection for energy-aware computing. In *IEEE International Conference on Cluster Computing*, October 2008.

[100] James D. Teresco, Jamal Faik, and Joseph E. Flaherty. Resource-aware scientific computation on a heterogeneous cluster. *Computing in Science & Engineering*, 7(2):40–50, 2005.

[101] TOP500.org. Top500 list – June 2009. http://www.top500.org/list/2009/06/, June 2009. Accessed on April 8, 2013.

[102] TOP500.org. Top500 list – November 2012. http://www.top500.org/list/2012/11/, November 2012. Accessed on March 23, 2013.

[103] Haluk Topcuouglu, Salim Hariri, and Min-you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, March 2002.

[104] Ten H. Tzen and Lionel M. Ni. Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):87–98, January 1993.

[105] Guibin Wang and Xiaoguang Ren. Power-efficient work distribution method for CPU-GPU heterogeneous system. In *International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, September 2010.

[106] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced CPU energy. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, November 1994.

[107] B. P. Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962.

[108] Changyou Zhang, Kun Huang, Xiang Cui, and Yifeng Chen. Programming-level power measurement for GPU clusters. In *IEEE/ACM International Conference on Green Computing and Communications (GreenCom)*, August 2011.

[109] Ying Zhang, Yue Hu, Bin Li, and Lu Peng. Performance and power analysis of ATI GPU: A statistical approach. In *International Conference on Networking, Architecture and Storage (NAS)*, July 2011.

[110] Gengbin Zheng, Michael S. Breitenfeld, Hari Govind, and Laxmikant V. Kalé. Automatic dynamic load balancing for a crack propagation application. Technical Report 06-08, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, June 2006.