**GPGPU MULTITASKING AND SCHEDULING**

By

Jacob T. Adriaens

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Electrical Engineering)

at the

UNIVERSITY OF WISCONSIN–MADISON

2012

Date of final oral examination: 05/31/12

The dissertation is approved by the following members of the Final Oral Committee:

Remzi H. Arpaci-Dusseau, Professor, Computer Sciences

Bradford M. Beckmann, Senior Design Engineer, AMD

Katherine Compton, Associate Professor, Electrical and Computer Engineering

Nam Sung Kim, Assistant Professor, Electrical and Computer Engineering

Mikko H. Lipasti, Philip Dunham Reed Professor, Electrical and Computer Engineering

Michael J. Schulte, Associate Professor, Electrical and Computer Engineering

## ACKNOWLEDGMENTS

A lot of contributions from others, direct and indirect, have been made to this work. I would like to thank those contributors here. I received funding first from the University of Wisconsin as a teaching assistant and later as a research assistant from the National Science Foundation (NSF) and the Semiconductor Research Corporation (SRC). AMD and NVIDIA have both donated GPUs used in this research. Tor Aamodt and his research group wrote GPGPU-Sim; the simulator used in this research. Without GPGPU-Sim I would have needed to write a new GPGPU simulator from scratch, both a difficult and time-consuming task.

I first became interested in GPUs as a research topic while I was an intern at AMD. Brad Beckmann was a fantastic host while I was at AMD. He was both welcoming and provided a wealth of knowledge on GPUs and computer architecture in general. He was also instrumental in refining the initial proposal of this research. I also met Andy Kegel while at AMD. In addition to educating me on many obscure areas of computer architecture, computer history, and hiking, Andy was largely responsible for solidifying the discussion of GPU memory virtualization in this document. AMD had a fantastic group of people I worked with and that experience solidified my decision to enter industry rather than academia after graduating. I started working full-time at Google approximately one year before completing my Ph.D. I greatly appreciate the support of my co-workers and Google's flexibility in allowing me to work full-time while I completed my Ph.D.

I initially decided to pursue a Ph.D. after William Hitchon suggested it during a lecture near the end of my undergraduate degree. Until that point in time I had never considered graduate school. Remzi Arpaci-Dusseau taught one of my early graduate courses. Remzi showed me the value of scientific publications, taught me how to critically analyze these publications, and showed me that research publications can actually be interesting.

I was advised by Seapahn Megerian while I obtained a Masters degree and during the beginning of my Ph.D. studies. Seapahn encouraged me to pursue many interesting research projects and I published my first paper with him. Seapahn taught me many important skills for being an effective researcher and encouraged me to continue graduate school even when it did not seem like the best idea to me. Through guidance and encouragement to build things rather than just think about them, he also helped refine the "engineer/hacker" mentality I already possessed into a useful research tool and an employable skill. To date Seapahn has had the largest contribution of any mentor to my success.

Michael Schulte and Katherine Compton served as my advisors for the latter half of my Ph.D. They have both been extraordinarily accommodating of family constraints, seem to tolerate extreme burstiness in my output, provide guidance when I need it and leave me to direct myself when I do not. In the academic world I have noticed that some advisors forget students are people too. Mike and Kati have never forgotten this. They have made possible the seemingly impossible task of graduating while working 260 miles from the University and then later holding a full-time job all while raising newborn twins.

Dan Gibson has been an excellent friend and collaborator throughout my graduate career. We have worked on many projects together and spent many days and nights discussing computers. He is an excellent technical resource; I bounce many half-formed ideas off of him. Our constant discussions of new ideas for hardware and software continue to keep me interested in the field.

My parents nurtured my interest in science and engineering from an early age. They taught me critical thinking and logic, took me to the library, science museum, and planetarium, bought me books on dinosaurs and astronomy, and also provided my first computer. This first computer cemented my future career path. They kept me in school and sent me to college at a time when I would not have continued on my own, and although I believe I

still would have succeeded in life, the path would have been far more difficult and far less certain. My parents also taught me how to build, repair, and figure out just about anything. These have turned out to be very useful skills and combined with the common sense and hard work ethic they gave me have propelled me to be quite successful.

The biggest sacrifices during graduate school have been made by my wife. TA and RA pay cannot support a family, so she has worked during most of my graduate career to support us. She has tolerated odd working hours, the strange friends you meet in graduate school, and long (apparent) lapses in work. While I interned at AMD we lived across the country from each other. Despite working full-time she cared for our kids and did almost all of the cooking, cleaning and shopping so I had more time for school. Without her continued support and sometimes aggressive encouragement for me to graduate I could not have done this. She deserves the Ph.D. more than I do and words cannot express how much I appreciate her sacrifices. I do not think this is a debt I will ever be able to repay.

# CONTENTS

Contents iv

List of Tables vi

List of Figures vii

Abstract ix

**1** Introduction 1
- *1.1 Motivation* 1
- *1.2 Dissertation Contributions* 3
- *1.3 Dissertation Structure* 4

**2** Background and Related Work 6
- *2.1 GPGPU, CUDA, and OpenCL* 6
- *2.2 GPU Architecture* 8
- *2.3 GPU Multitasking* 13
- *2.4 GPGPU* 16
- *2.5 GPU Simulation and Evaluation* 17
- *2.6 Multitasking and Scheduling* 19

**3** Methodology 26
- *3.1 Application Descriptions* 26
- *3.2 Simulation Environment* 30
- *3.3 Simulation Length* 36

**4** Application Characterization 38
- *4.1 Methodology* 38
- *4.2 Evaluation* 38
- *4.3 Conclusion* 43

**5** Spatial Multitasking 44
- *5.1 Methodology* 44
- *5.2 Evaluation* 45
- *5.3 Implementation Details* 52
- *5.4 Conclusion* 68

**6** SM Partitioning 69
*6.1 Methodology* 69
*6.2 Evaluation* 75
*6.3 Conclusion* 80

**7** Quality-of-Service 82
*7.1 Evaluation* 82
*7.2 Conclusion* 90

**8** Recent Developments, Future Research, and Conclusions 91
*8.1 Recent Developments and Future Research* 91
*8.2 Conclusions* 93

**A** Derivation of Application Execution Times in Isolation 96

**B** Proof of Wait is Better Than Terminate and Migrate 101

References 105

## LIST OF TABLES

2.1 List of common abbreviations. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

3.1 Workload types. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

3.2 Application kernel configuration. . . . . . . . . . . . . . . . . . . . . . . . . . . 28

3.3 Baseline GPGPU-Sim configuration, NVIDIA Quadro FX 5800. . . . . . . . . . . 30

5.1 Even split speedup of spatial multitasking compared to cooperative multitasking. 47

5.2 Spatial multitasking speedup for several benchmark combinations. . . . . . . . 47

5.3 Summary of DRAM bandwidth utilization for all combinations of applications. 48

5.4 Summary of average interconnect latency for all combinations of applications. . 49

5.5 Global memory footprint of applications. . . . . . . . . . . . . . . . . . . . . . . 53

5.6 Observed GPGPU kernel runtimes. . . . . . . . . . . . . . . . . . . . . . . . . . 54

6.1 Average kernel thread configuration. . . . . . . . . . . . . . . . . . . . . . . . . 73

6.2 Spatial multitasking speedup by partitioning heuristic. . . . . . . . . . . . . . . 76

6.3 Distribution of SMs for several specific combinations of applications. . . . . . . 79

6.4 Speedup of spatial multitasking over cooperative multitasking for several specific combinations of applications. . . . . . . . . . . . . . . . . . . . . . . . . . 80

7.1 Spatial multitasking QoS speedup by heuristic. . . . . . . . . . . . . . . . . . . . 85

7.2 Spatial multitasking QoS performance degradation. . . . . . . . . . . . . . . . . 87

7.3 Speedup of dynamic QoS partitioning compared to static. . . . . . . . . . . . . . 88

## LIST OF FIGURES

2.1 Block diagram of the GT200 architecture. . . . . 9
2.2 Illustration of multitasking methods. . . . . 15

3.1 Comparison of CUDA, GPGPU-Sim and GPGPU-Sim-multitasking. . . . . 33
3.2 Example GPGPU-Sim-multitasking configuration file. . . . . 34
3.3 General GPGPU-Sim-multitasking configuration file format. . . . . 35
3.4 GPGPU-Sim-multitasking scheduler interface. . . . . 36
3.5 Spatial multitasking speedup versus simulation length. . . . . 37

4.1 Speedup of GPU computation versus number of SMs. . . . . 40
4.2 Speedup of GPU computation versus GPU memory frequency. . . . . 41
4.3 Speedup of GPU computation versus GPU interconnect frequency. . . . . 42

5.1 Distribution of spatial multitasking speedup. . . . . 46
5.2 CDF of DRAM bandwidth utilization under spatial multitasking. . . . . 48
5.3 CDF of average interconnect latency under spatial multitasking. . . . . 49
5.4 Average GPU interconnect latency versus number of SMs. . . . . 50
5.5 Comparison of performance scaling in isolation and spatial multitasking. . . . . 51
5.6 Thread migration timing, case 1. . . . . 59
5.7 Thread migration timing, case 2. . . . . 59
5.8 Thread migration cost versus remaining execution time. . . . . 62

6.1 Speedup of GPU computation versus number of SMs. . . . . 71
6.2 Spatial multitasking speedup by partitioning heuristic. . . . . 77
6.3 Two application spatial multitasking speedup versus number of SMs by partitioning heuristic. . . . . 78

6.4 Three application spatial multitasking speedup versus number of SMs by partitioning heuristic. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79

7.1 Spatial multitasking QoS speedup by partitioning heuristic. . . . . . . . . . . . 84
7.2 Distribution of SAD QoS performance. . . . . . . . . . . . . . . . . . . . . . . . . 86
7.3 Performance of RSA when sharing the GPU via spatial multitasking. . . . . . . 89

A.1 Thread migration timing, case 1. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97
A.2 Thread migration timing, case 2. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97

## ABSTRACT


The set-top and portable device market continues to grow, as does the demand for improved performance, cost, and power. The integration of *Graphics Processing Units* (GPUs) into these devices and the emergence of general-purpose computations on graphics hardware enable a new set of highly parallel applications. To facilitate these applications, I propose and make the case for a novel GPU multitasking technique called *spatial multitasking*. Traditional GPU multitasking techniques, such as cooperative and preemptive multitasking, partition GPU *time* among applications, while spatial multitasking allows GPU *resources* to be partitioned among multiple applications simultaneously. I demonstrate the potential benefits of spatial multitasking with an analysis and characterization of *General-Purpose GPU* (GPGPU) applications.

My analysis indicates that many GPGPU applications fail to utilize available GPU resources fully, which suggests the potential for significant performance benefits using spatial multitasking instead of, or in combination with, preemptive or cooperative multitasking. I use simulation to evaluate a simple implementation of spatial multitasking against cooperative multitasking. This implementation of spatial multitasking shows an average speedup of 1.14, 1.22, and 1.30 compared to cooperative multitasking when two, three, and four applications share the GPU, respectively. I follow this with an evaluation of more complex resource partitioning heuristics for spatial multitasking and show even greater speedup of spatial multitasking over cooperative multitasking. Finally, I explore providing Quality-of-Service to GPGPU applications using spatial multitasking and show significant benefits compared to Quality-of-Service using temporal multitasking.

# 1 INTRODUCTION

Set-top and portable devices are becoming increasingly popular and powerful. Due to the cost, power, and thermal constraints placed on these devices, often they are designed with a low-power general-purpose CPU and several heterogeneous processors, each specialized for a subset of the device's tasks. These heterogeneous systems increasingly include programmable *Graphics Processing Units* (GPUs). The iPhone 3GS, for example, contains a programmable GPU in addition to a general-purpose CPU and several *Application Specific Instruction Processors* (ASIPs) [1]. Transforming the GPU from a graphics-only device to a general-purpose data-parallel processor has the potential to enable entirely new classes of applications that were previously unavailable to mobile devices due to performance and power constraints.

GPGPU computations are motivated by GPUs' tremendous computational capabilities and high memory bandwidth for data-parallel workloads [2]. A range of applications, from scientific computing and multimedia processing, are well-suited to this form of parallelism and achieve large speedups on a GPU compared to a CPU. For example, Yang *et al.* achieve up to a 38x speedup compared to a high-performance CPU when using a GPU for real-time motion estimation [3].

## 1.1 Motivation

Unfortunately, GPUs have very primitive support for multitasking, a key feature of modern computing systems. Multitasking provides the illusion of concurrent execution of multiple applications on a single device. Improving GPU multitasking is critical for preserving user responsiveness and satisfying *quality-of-service* (QoS) requirements. Other applications needing the GPU must wait until the application occupying the GPU voluntarily yields

control. This form of multitasking is called *cooperative multitasking*. In contrast, on the CPU, the operating system (OS) typically uses *preemptive multitasking* on a single CPU core—suspending and later resuming applications to time-share the CPU core without the applications' intervention or control.

Applications can also space-share resources, executing in parallel on subsets of resources instead of time-sharing the full set. I refer to this form of multitasking as *spatial multitasking*. To be specific, spatial multitasking differs from temporal (cooperative or preemptive) multitasking in that it divides *resources*, rather than *time*, among applications. Parallel CPU-based systems, from multicore CPUs to massively parallel supercomputers, typically use spatial multitasking to allow multiple threads to simultaneously execute. Currently, spatial multitasking allows multiple applications to execute across multiple GPUs; however, it does not as of yet permit multiple applications to execute simultaneously within a single GPU's significantly parallel resources. Instead, GPUs support cooperative multitasking, which suffers from poor application response time, uses GPU resources inefficiently, and is open to security and reliability issues. NVIDIA's Fermi GPU represents a step towards spatial multitasking in that it supports co-executing multiple *tasks* from the *same* application on a single GPU [4]. Unfortunately, it does not support co-executing tasks from *different* applications.

In my work, I demonstrate that many GPGPU applications fail to fully utilize GPU resources, such as memory bandwidth and GPU cores, called *stream multiprocessors* (SMs). With future GPUs providing even more resources, additional applications are likely to underutilize GPUs. Spatial multitasking improves total system performance over preemptive and cooperative multitasking by more fully utilizing GPU resources. Spatial multitasking more efficiently uses the GPU by allowing multiple applications to execute on the GPU simultaneously, thereby sharing the resources and reducing context switch overhead. I demonstrate this with a characterization of isolated application utilization

of several GPU resources and evaluate an implementation of spatial multitasking in a cycle-accurate execution-driven GPU simulator and compare it to cooperative multitasking.

Until GPUs better support multitasking, they will continue to remain second-class computational citizens. As future technologies move the GPU onto the same chip as the CPU [5], the importance of advancing the GPU from a graphics-only coprocessor to a full-fledged multitasking parallel accelerator will grow. This will require development of new GPU multitasking techniques, both temporal and spatial.

## 1.2 Dissertation Contributions

This document presents a characterization of GPGPU applications for the portable and set-top markets. Using this characterization, I demonstrate significant potential performance benefits for spatial multitasking compared to preemptive and cooperative multitasking. I then evaluate a simple implementation of spatial multitasking and compare it to cooperative multitasking, confirming the performance potential of spatial multitasking. I then explore the architectural and system challenges of implementing spatial multitasking. The simple evaluation of spatial multitasking is followed by an evaluation of SM partitioning heuristics and an analysis of QoS in spatial multitasking. The key contributions of this work are:

- The proposal of GPGPU spatial multitasking, a novel technique for GPUs in which applications execute simultaneously with SMs partitioned among them, rather than execute serially on all GPU resources. GPGPU spatial multitasking is similar to multitasking on multicore CPUs.
- A detailed characterization of GPGPU applications in which I show many GPGPU workloads underutilize GPU resources when executing in isolation on the GPU.

- An evaluation of GPGPU spatial multitasking versus cooperative multitasking through cycle-accurate simulation.
- An analysis of system and architectural challenges that must be addressed when implementing spatial multitasking.
- A discussion of thread migration strategies on GPU SMs in spatial multitasking.
- A comparison of several GPU SM partitioning heuristics for spatial multitasking.
- An evaluation of QoS-aware partitioning using spatial multitasking.
- Extensions to a GPU simulator for simulating temporal and spatial multitasking.

Most existing GPGPU research has focused on application development and acceleration, while this research focuses on architecture and system-level GPU improvements. I examine modifying the GPU system software to raise GPU utilization, improve application response time, and increase throughput of GPGPU applications through improved multitasking and scheduling. This research aims to help transform GPUs into general-purpose data-parallel processors by adding modern multitasking capabilities. This enables new multimedia and other data-parallel applications on set-top and portable devices by improving total system performance when multiple applications share the GPU.

## 1.3 Dissertation Structure

This document is organized as follows. Chapter 2 provides background on GPGPU computing, GPU architecture, the current state of GPU multitasking, and a discussion of related work. Chapter 3 presents the methodology used. This includes a description of the applications used in this work, the simulation environment, and the simulator changes necessary to evaluate spatial multitasking. My evaluation of spatial multitasking starts

in Chapter 4 with a characterization of application performance scaling when running in isolation. This characterization provided the initial insight that spatial multitasking is likely to more efficiently use GPGPU resources over cooperative multitasking. Chapter 5 evaluates spatial multitasking in simulation versus cooperative multitasking and discuss the hardware and system software challenges that must be addressed when implementing spatial multitasking. Chapter 5 evaluates spatial multitasking when GPU cores are split evenly among applications sharing the GPU, in Chapter 6 I evaluate more complex partitioning heuristics. Chapter 7 explores providing Quality-of-Service to applications when using spatial multitasking. Finally, Chapter 8 summarizes the results of my research and highlights potential areas for future work.

## 2.1 GPGPU, CUDA, and OpenCL

Early GPGPU computation on commodity desktop GPUs started with fixed-function graphics pipelines programmed through graphics APIs, such as DirectX and OpenGL. Using fixed-function hardware greatly limited the computations that could be performed and using the graphics API required the programmer to transform their data into an image or part of a scene to be rendered, then convert the data back after GPU processing. In 1999, NVIDIA introduced the first programmable GPU with limited support for programmability in the fragment stage of the rendering pipeline [6]. In 2002, ATI transitioned the programmable GPU from fixed-point to both fixed-point and floating-point [7]. The move from a fixed to a programmable graphics rendering pipeline, coupled with the development of domain-specific "shader languages", allowed programmers to use the GPU to perform a much wider variety of computations. The shader languages were initially similar to assembly, though later were based on C. Examples include OpenGL GLSL and DirectX HLSL. In late 2006, AMD announced *Close To the Metal* (CTM) [8] and NVIDIA announced CUDA [9], both major advances that allowed the GPU to be programmed directly rather than through a graphics API.

Although a number of APIs for the GPU now exist, for this research I use NVIDIA's CUDA due to the greater availability of applications written for it over other APIs. In the future, OpenCL may eclipse CUDA. While CUDA is specific to NVIDIA GPUs, a number of companies support OpenCL, including AMD, Apple, Intel and NVIDIA. Applications written for OpenCL are intended to run efficiently on various parallel platforms including the CPU, the GPU, or both. Although this research uses the CUDA API, the results should generalize to a number of other GPGPU APIs including OpenCL.

CUDA programs are written in a subset of C++ with CUDA-specific extensions, this language is called "C for CUDA." Programmers must explicitly split code into GPU and CPU portions and must identify GPU functions for the C/C++ preprocessor. In addition to specifying which code is for GPU execution and which code is for CPU execution, the programmer must also explicitly allocate and deallocate memory on the GPU separately from the CPU. There are also special functions for copying data between CPU and GPU memory. In CUDA, functions that are off-loaded to the GPU are called *kernels*. When calling a kernel, the programmer must specify the number of thread blocks and threads per block to execute the kernel with. *Thread blocks* are a group of threads that share a processor on the GPU and are able to communicate through fast shared memory. Communication is possible between thread blocks through a slower global memory. Each thread block and thread is given a unique identifier that can be read at runtime. Control flow and data can diverge based on this identifier allowing unique computation across threads and thread blocks. Threads and thread blocks are discussed further in Section 2.2. A CUDA kernel executes asynchronously on the GPU; the CPU thread that launched the kernel may continue execution in parallel. CUDA provides several synchronization functions to allow the CPU thread to wait for data from the GPU.

CUDA compilation process takes several steps. First, the source code is preprocessed by the *Edison Design Group* (EDG) C++ compiler [10]. The preprocessor splits the source code into new files, separating the CPU code from the GPU code. The CPU code is then compiled with the GNU C/C++ compiler. The GPU code is compiled with a customized version of the Open64 compiler. The Open64 compiler outputs assembly targeting a virtual ISA for NVIDIA GPUs called *Parallel Thread eXecution* (PTX). The GPU does not execute PTX code directly, instead when the PTX code is run the graphics driver translates it into GPU-specific instructions. PTX is discussed further in Section 3.2.

Table 2.1: List of common abbreviations.

| Full Name | Abbreviation |
|---|---|
| Graphics Processing Unit | GPU |
| General Purpose GPU | GPGPU |
| Quality-of-Service | QoS |
| Single Instruction Multiple Data | SIMD |
| Single Instruction Multiple Thread | SIMT |
| Stream Multiprocessor | SM |
| Thread Processing Cluster | TPC |
| Execution Unit | EU |
| Special Functional Unit | SFU |
| Load Store Unit | LSU |
| Floating Point Unit | FPU |

## 2.2 GPU Architecture

This research uses the NVIDIA Quadro FX 5800 GPU as a baseline architecture. The Quadro FX 5800 was chosen because it was a state of the art high performance GPU with a large number of processors and it supports CUDA. It is architected similarly to other recent GPUs, so research using the Quadro FX 5800 should generalize well to other GPUs. The Quadro FX 5800 is an implementation of NVIDIA's GT200 architecture, which is depicted in Figure 2.1. Although NVIDIA has published a general overview of the architecture [11], many details of the architecture are not public. Therefore many of the details contained in this section are speculative based on CUDA documentation, older GPU architectures and rumored NVIDIA leaks. The primary source of speculated details is a web article by David Kanter [12]. Table 2.1 summarizes some common GPU-related abbreviations.

The GT200 architecture has 10 *thread processing clusters* (TPCs) each made up of 3 *stream multiprocessors* (SMs), for a total of 30 SMs. The SMs support both 32-bit and 64-bit floating-point operations. When performing 32-bit floating-point operations, each SM acts as an 8-wide SIMD multiprocessor. This allows the GPU to execute up to 240 threads simultaneously.

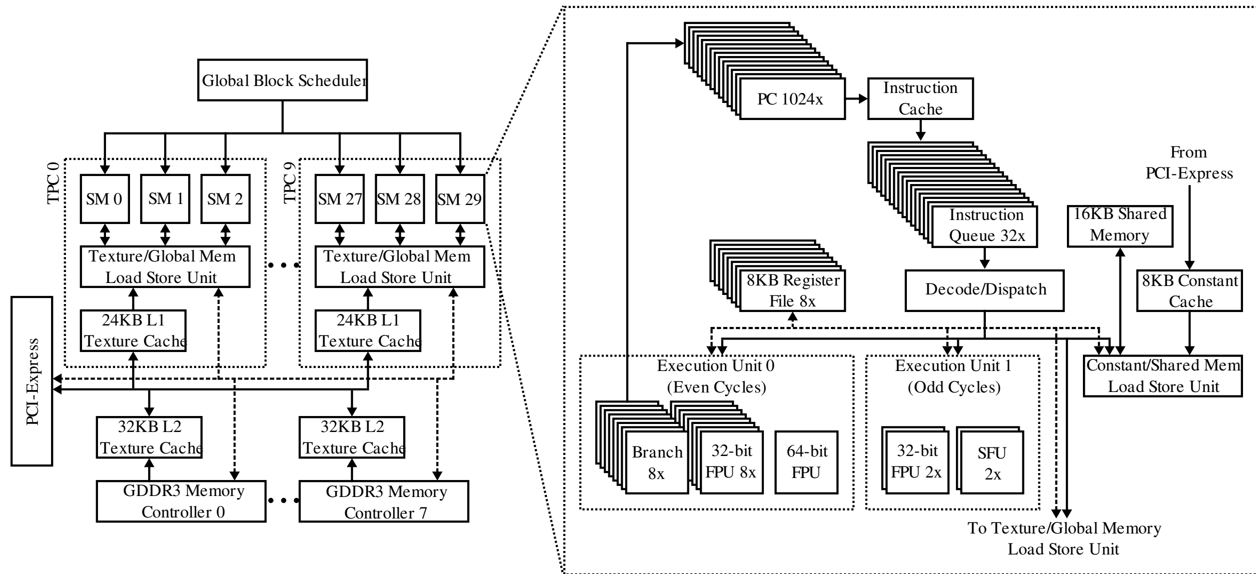



Figure 2.1: Block diagram of the GT200 architecture [12].

An SM is slightly different from a traditional *single instruction multiple data* (SIMD) processor. A traditional 8-wide SIMD processor has a single PC per thread and operates on 8-wide data vectors, while an SM has 8 PCs per thread. This allows the programmer to write software for the GPU using threads rather than vectors. Control flow divergence is handled in hardware and is transparent to the programmer. In traditional SIMD code, the programmer or compiler must explicitly use vector masks for predicated execution when different operations are performed on different parts of the vector. On the other hand a GPU SM automatically masks off the writeback of results from threads not following the current execution path. NVIDIA calls this type of processing *single instruction multiple thread* (SIMT) to distinguish it from SIMD. When control flow among the 8 threads executing in an SM diverges, the dispatcher serially dispatches instructions from each execution path using predicated execution until control flow reconverges. Therefore, having $N$ execution paths for each of the 8 threads that dispatch together results in a factor of $N$ slowdown. NVIDIA does not disclose how the dispatch logic chooses which execution path to dispatch

from, but this can have a large impact on performance because it determines how threads reconverge. Fung *et al.* have proposed several heuristics for instruction scheduling to aid in thread reconvergence [13].

As shown in Figure 2.1, each SM actually has two *execution units* (EUs) of differing capabilities to which it can dispatch. EU0 can receive instructions on even cycles and EU1 on odd cycles[1]. EU0 supports 8-wide SIMD for 32-bit floating-point operations and branch resolution, but does not implement SIMD operations for 64-bit floating-point operations. EU1 does not execute 64-bit floating-point operations and only supports 2-wide SIMD operations. The 32-bit *floating-point units* (FPUs) of EU0 execute fused multiply-add in 4 cycles. The FPUs of EU1 do not implement fused multiply-add and therefore require separate multiply and add instructions for a total latency of 8 cycles. Unlike a fused multiply-add, this pair of instructions rounds between the multiply and the add, reducing accuracy in floating-point computations. The final difference between EU0 and EU1 is the 2-wide SIMD *special functional unit* (SFU) in EU1. The SFU is primarily for graphics applications and supports complex operations such as sine and cosine. These operations typically take 16 cycles or more to execute. Each SM has 8 register files, one for each thread that can execute during a single cycle. Each register file has 2K 32-bit entries. All arithmetic operations in an SM are register-to-register.

There are three clock domains in the GT200 architecture: GPU, shader and global memory. FIFOs are placed between clock domains for buffering. The GPU clock domain covers all of the GPU, except the execution units of the SMs and the off-chip DRAM. The SM execution units are in the shader clock domain, and the off-chip DRAM is in the global memory domain.

[1]The dispatch unit and the EUs are actually in different clock domains, so an EU typically executes a new instruction every 4 cycles in its own clock domain, which is roughly every 2 cycles in the dispatcher's clock domain. This causes EU0 to effectively be a 32-wide SIMD because the same instruction is executed (on different threads) for each of the 4 cycles on the 8-wide SIMD unit.

In the GT200 architecture threads are organized into *blocks* and *warps*. A thread block is a group of up to 512 threads that are scheduled to a single SM, start from a single PC, and are able to share data through a small local memory. A warp is a subset of a thread block whose threads execute together in SIMT fashion. This subset is composed of 32 threads that issue instructions and memory requests together. If 32 threads are not available to fill a warp, empty threads fill the remainder. The write-back of empty threads is masked off. A warp is 32 threads because of the effective width of 32 (4 cycles x 8-wide SIMD) for the SIMD execution units. Although warps are not architecturally visible to the programmer, the programmer should be aware of them to achieve good performance. Specifically, control flow divergence should be avoided within a warp because each divergent path must be executed serially rather than in parallel.

The global block scheduler schedules thread blocks onto SMs. Each SM is capable of having up to 8 thread blocks and up to 1024 threads scheduled to it, whichever is the limiting factor. SMs perform *interleaved multithreading* (IMT) with up to 1024 threads to hide long latency events. Each SM has an instruction cache, however the details of this cache remain undisclosed. There are also 32 instruction queues, one for each of the warps that can run on an SM. Instructions are dispatched once all the operands are ready for all 32 threads in the warp. An instruction typically takes 4 cycles to execute, with 8 threads from the warp executing each cycle. During long latency events, such as a load, it is easy to keep the execution units in the SM busy because there are up to 32 warps from which to dispatch. Although there is no speculative execution in an SM, there can be out of order execution if a long latency event, such as a load, issues and is followed by arithmetic or branch instructions that are not dependent on that event.

The GT200 architecture has four logical memories, some of which share physical hardware. The first two are the *global memory* and *texture memory*. Both share the same physical set of ten *load store units* (LSUs), interconnect, memory controllers and off-chip GDDR3

DRAM, as shown in Figure 2.1. A single texture and global memory LSU is shared by the three SMs within a TPC. An SM simultaneously issues loads and stores for all 32 threads in a warp to the LSU. The LSU will attempt to coalesce several localized memory operations, possibly from other thread blocks, into one memory operation for increased memory performance. The eight memory controllers connect to DRAM via a 512-bit GDDR3 bus. Both texture and global memory use 40-bit virtual addressing. Details of the texture and global memory interconnect are undisclosed. However, Bakhoda *et al.* show that CUDA applications are relatively insensitive to interconnect latency and bandwidth if the bandwidth surpasses a minimum threshold [14]. Although the global and texture memories share hardware, they are logically separated and accessed differently by the programmer.

Global memory is shared by all threads on the GPU. It is an uncached off-chip GDDR3 DRAM that is both readable and writable from the GPU. Because global memory is uncached, it suffers from high access latency and low bandwidth compared to on-chip memories, but is considerably larger. Global memory supports atomic operations for sharing data between arbitrary threads on the GPU. Global memory can also be read and written by the CPU over the PCI-Express bus. The purpose of global memory is to handle overflow (in software) of data from the register file and other memories. It also facilitates data sharing between thread blocks.

Texture memory is read-only in GPGPU applications and is loaded by the CPU via PCI-Express before computation on the GPU begins. Unlike global memory, there are two levels of cache in the texture memory hierarchy. L1 caches are 24KB in size and are shared by three SMs grouped in a TPC. Texture cache lines are 64B, but unlike CPU caches, the texture cache has 2D locality to optimize for graphics applications. In other words, when a piece of data misses in a texture cache a 4x4 square of data is brought in rather than 16 elements in a line in memory. Each memory controller has a 32KB L2 texture cache. The replacement policy and associativity of the texture caches are undisclosed. The latency

of the texture caches is quite high; the texture caches primarily serve to reduce off-chip memory bandwidth. Although in GPGPU applications the texture cache is read-only from the perspective of the GPU, it is writable by the CPU and writable by the GPU in graphics applications. The texture caches have a very simple coherency mechanism; when the CPU or a graphics application writes a texture all of the texture caches are invalidated.

The third memory is called the *constant cache*. Despite the name it is actually an 8KB read-only memory within each SM. Like texture memory, the constant caches are loaded by the CPU before a program executes on the GPU. However, unlike texture memory, the constant cache is actually just an on-chip memory and is not backed by higher level caches or an off-chip memory.

Finally, the GT200 *shared memory* is a 16KB on-chip memory local to each SM. The shared memory is low latency, high bandwidth and is used to share data only between threads within a thread block; global memory must be used to share memory across thread blocks. Once a thread block is assigned to an SM it never migrates to another SM, so there is no support for exchanging data directly between shared memories. Unlike the other memories, the CPU cannot access shared memory. Like global memory, shared memory supports atomic operations. GPGPU programs also have a barrier/memory fence instruction available to facilitate proper global and shared memory ordering within a thread block.

## 2.3 GPU Multitasking

Initially, multiple graphics applications could only share a GPU via *cooperative multitasking*, requiring an application executing on the GPU to yield GPU control voluntarily. If a malicious or malfunctioning application never yielded, other applications were unable to use the GPU. Windows Vista, together with DirectX 10, introduced GPU *preemptive*

*multitasking* for graphics applications, but not for GPGPU applications [15, 16]. In Windows Vista and Windows 7, GPGPU applications still use cooperative multitasking. In cooperative multitasking, a computation block off-loaded to the GPU runs to completion before yielding the GPU. NVIDIA refers to this computation block as a *kernel*. A GPGPU application may contain one or more kernels. To avoid problems with malfunctioning and malicious GPGPU applications, Windows Vista and Windows 7 impose time limits on GPU computations, after which the OS requests that applications yield the GPU. If an application fails to yield, the GPU is reset, killing any active GPU computations [17]. Thus, GPGPU applications must be coded explicitly to yield the GPU during long computations so they will not be terminated. This means breaking up long GPGPU computations into a sequence of shorter computations. Further complicating the issue, different GPUs have varying performance characteristics, so computations that complete in the allotted time on one GPU may not on another. Furthermore, even within the time limits, GPGPU calculations may be quite long, sacrificing interactive response time. Preempting applications from the GPU and/or allowing applications to run simultaneously could help solve these issues.

Although preemption addresses some GPU multitasking issues, there is a large overhead associated with context switches: saving the current GPGPU state of one application and restoring another's. This state includes the register file and the GPU cores' local memory data. For example, in the NVIDIA GT200 architecture, each SM has a 64KB register file, 8KB constant cache, and a 16KB shared memory. A kernel using all 30 SMs of this architecture has a state size greater than 2.5MB [12]. In contrast, an AMD64 CPU core has 128 bytes of general-purpose registers, 256 bytes of media registers, and 80 bytes of floating-point registers [18]; this and other state data together represent approximately 0.5KB that must be saved and restored for an AMD64 CPU context switch. The larger GPU kernel context size results in significantly more overhead for a GPU context switch than a context switch on the CPU.

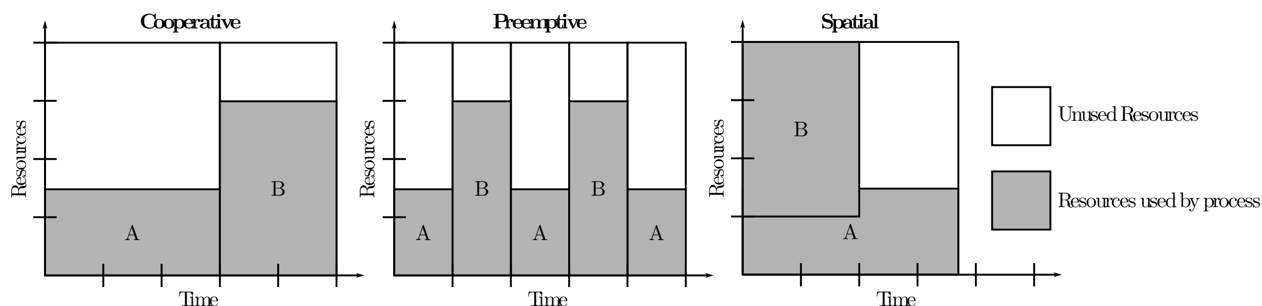



Figure 2.2: Illustration of multitasking methods. A and B represent applications using the GPU. The Y-axis represents the GPU resources and the X-axis represents the duration those resources are used. In spatial multitasking, the applications execute simultaneously with resources split among them.

To address the problems and challenges associated with temporal multitasking, I propose *spatial multitasking*—allowing multiple GPGPU kernels to execute simultaneously, each using a subset of the GPU resources. Spatial multitasking differs from preemptive multitasking in that it divides GPU *resources*, rather than GPU *time*, among competing applications. For example, instead of giving two applications 100% of the GPU resources 50% of the time, spatial multitasking could grant each application 50% of the GPU resources 100% of the time. If one application completes, the other could then use 100% of the GPU resources. Figure 2.2 illustrates the differences between cooperative, preemptive, and spatial multitasking.

I have observed that many GPGPU workloads are tuned for a particular GPU generation and subsequent, more aggressive GPUs frequently are underutilized by software just one generation old. Spatial multitasking can increase GPU utilization; in combination with temporal multitasking, it should improve system performance compared to temporal multitasking alone. Dividing resources among applications will also reduce context switches, further improving performance.

This research explores the performance advantages of GPGPU spatial multitasking over cooperative multitasking. I show applications running in isolation often do not

fully utilize GPU resources. By splitting GPU SMs among multiple applications and executing in parallel I demonstrate spatial multitasking achieves higher GPU utilization and consequently improves total system throughput when compared to serial execution via cooperative multitasking. I also explore a number of heuristics for splitting SMs among applications ans examine QoS under spatial multitasking.

## 2.4 GPGPU

Non-graphics computation on commodity graphics hardware began with fixed-function GPUs [6]. This early work formulated general-purpose problems so they could be accelerated by a portion of the non-programmable GPU graphics hardware such as the rasterizer and z-buffer [19]. Later GPUs became programmable through the vertex and pixel shaders. Vertex shading is the first step in a traditional real-time graphics rendering pipeline. A series of vertices represent the scene to be rendered and the vertex shader modifies the position of these vertices. Pixel shading is the last stage in the traditional real-time graphics rendering pipeline. A pixel shader takes a rasterized image and adjusts the color of each pixel. GPGPU applications targeting this hardware [20] were still restricted to using the graphics pipeline through graphics APIs, such as OpenGL, but were given more flexibility through limited programmability. Programs were limited to graphics-specific shader languages such as GLSL. The next step in GPU evolution involved combining the vertex and pixel shaders into a single programmable set of SIMD processors. In conjunction with this change NVIDIA and AMD released tool-chains for their GPUs which allowed programmers to directly program the SIMD processors using C-like languages, and bypass the graphics software pipeline entirely.

A vast amount of research has focused on accelerating algorithms and applications using these modern GPUs. Some examples include:

- Data clustering [21]
- Neural network simulation [22]
- Detection and tracking of white blood cells in video microscopy [23]
- Monte Carlo simulation [24]
- Ultrasound simulation [25]
- On-chip power grid analysis [26]
- Ray tracing based rendering [27]
- AES Encryption [28]
- JPEG encoding and decoding [29]
- Fractal generation [30]
- Video encoding and decoding [31]

While these examples focus on improving the performance of specific applications by accelerating them with the GPU, my work examines architectural and system-level multitasking changes to improve system throughput and GPU utilization when multiple GPGPU applications execute simultaneously.

## 2.5 GPU Simulation and Evaluation

Although a large amount of research has focused on accelerating applications and algorithms on the GPU, little published research has proposed GPU architectural changes to aid GPGPU applications. Consequently, very few tools have been developed for GPGPU

hardware exploration. For this research, I have chosen to extend GPGPU-Sim, which is discussed further in Section 3.2. However, several other GPU simulators exist. Qsilver is a trace-based GPU micro-architectural simulator [32, 33]. Qsilver provides a performance, power and thermal model for the GPU; however it is not useful for this research, as it is only able to simulate graphics applications and not GPGPU applications. ATTILA is a cycle-accurate execution-driven GPU simulator; like Qsilver, it only simulates graphics applications [34]. Barra is a functional GPU simulator capable of simulating unmodified CUDA binaries; unfortunately the lack of a performance model excludes its use for this research [35]. CUDA itself provides the ability to functionally simulate GPGPU applications, but this only is for debugging purposes. CUDA does not provide performance information when functionally simulating on the CPU and cannot be modified to simulate the multitasking modifications evaluated in this work because it is proprietary.

Even with a simulator that includes a performance model, there are challenges in measuring the performance of applications in a multitasking environment. Rupnow *et al.* explore the challenges of accurately and meaningfully evaluating performance in hybrid computing systems with multitasking capabilities [36]. They propose the *Normalized Equivalent Instruction Count* (NEIC) performance metric to address the following common problems faced when evaluating hybrid computing systems through full-system cycle-accurate simulation:

- Limited simulation time.
- Difficulty of comparing performance across heterogeneous processing elements.
- Multitasking scheduling variations that do not average out during short simulations.
- Measuring, eliminating variation, and properly attributing system overheads.

I use a simplified version of the NEIC metric in this research because I simulate a homogeneous system. However, many of the challenges discussed by Rupnow *et al.* are the same challenges I faced in this evaluation of spatial and cooperative multitasking on the GPU. Consequently, the work by Rupnow *et al.* has heavily influenced the methodology used throughout this work.

## 2.6 Multitasking and Scheduling

Multitasking and scheduling have been explored for a variety of platforms that share some properties with GPUs. NVIDIA's Fermi architecture supports a very limited form of GPU spatial sharing. Intel's Larrabee and the Cell multiprocessor are multitasking capable graphics-like architectures. Scheduling of both homogeneous and heterogeneous processing resources has been extensively studied. The issue of time-sharing versus space-sharing analyzed here in my research for GPUs has already been evaluated in the context of multicore CPUs. QoS scheduling of GPU applications has been explored using time multiplexing, in this research I evaluate QoS using spatial multitasking.

NVIDIA's Fermi architecture supports concurrent execution of GPGPU kernels from a single application [4]. Similar to spatial multitasking, this allows the GPU to be used more efficiently when there are multiple kernels available to run in parallel. However, unlike spatial multitasking, Fermi's support for concurrent execution of GPU kernels is limited to sets of kernels from the *same* application. The opportunity to execute multiple kernels from the same application in parallel is likely to be rare compared to the frequency of multitasking separate GPGPU applications, partly caused by limitations imposed by data dependencies across kernels, a problem not present in spatial multitasking. Our work demonstrates that Fermi and other GPU architectures would likely benefit from true spatial multitasking. Implementing spatial multitasking presents a number of challenges that

Fermi does not address, including but not limited to, scheduling and resource partitioning among applications and process isolation.

The Cell multiprocessor is made up of a general-purpose super-scalar processor and several SIMD processors [37]. The SIMD processors, known as *Synergistic Processing Elements* (SPEs), support both cooperative and preemptive multitasking, but not spatial multitasking. The context of an SPE is roughly 256KB [38]. Although this is large compared to a CPU, it is still much smaller than a GPU, making preemptive multitasking more attractive on Cell than on a GPU. However, spatial multitasking is still likely to improve performance in the Cell multiprocessor by using resources more efficiently. Like Larrabee, the scale of parallelism on the Cell multiprocessor is much smaller than what my work targets, which presents different challenges for multitasking.

Intel's Larrabee was proposed as a many-core architecture intended for graphics processing [39]. Although the Larrabee literature does not discuss multitasking, it is very likely that Larrabee would have fully supported preemptive and spatial multitasking for Larrabee's general purpose cores, like most other x86-based multicore CPUs. This work differs from Larrabee by investigating multitasking for standard GPUs, which have a large number of very simple processors. Conversely, Larrabee was to have a smaller number of general-purpose super-scalar processors extended with vector-processing capabilities. Additionally, the number of hardware threads supported on each GPU core is much higher than the four threads per processor supported on Larrabee; standard GPUs are optimized for a very large number of threads running on simple processors, while Larrabee had less parallelism available on fewer processors.

Intel's EXOCHI is an extension of the OpenMP API compiler and run-time environment that supports multi-ISA *Heterogeneous Chip Multiprocessors* (HCMPs) [40]. It allows programmers to specify regions of code to map to specific types of accelerators. The compiler produces a single fat binary with multiple ISAs embedded within it, and the run-time envi-

ronment automatically schedules parallel sections of code onto the programmer-specified accelerators. This allows a single program to run multiple threads simultaneously on heterogeneous processors. However, multitasking is not investigated. Scheduling is limited to mapping threads from a single application onto multiple accelerators, whereas scheduling in this research focuses on the mapping of multiple applications onto a single accelerator, the GPU.

Fu and Compton propose and analyze heuristics for dynamically distributing reconfigurable hardware resources among multiple software applications [41]. These applications execute on a general-purpose multicore CPU and use the reconfigurable hardware to implement accelerators. The problem of dynamically distributing a fixed number of reconfigurable logic blocks among applications is similar to the SM partitioning problem addressed in this research in Chapter 6.

The issue of time-sharing versus space-sharing has been investigated on shared-memory multiprocessors as well. McCann *et al.* showed space-sharing scheduling policies outperform time-sharing [42]. They concluded this is because parallel applications tend to have convex speedup-versus-resource curves, caused by insufficient parallelism and overheads that grow with the number of processors used. Although GPUs are architected quite differently from shared-memory multiprocessors, GPGPU applications still tend to have convex speedup-versus-resource curves, as I show in Chapter 4. This feature is one of the motivating factors for GPU spatial multitasking.

Ousterhout also compared time-sharing and space-sharing, and noted that applications that exhibit fine-grained inter-process communication perform poorly when only a subset of the communicating processes are executing concurrently [43]. Co-scheduling is proposed to handle this situation. In co-scheduling, processes from the same job are scheduled to multiple processors simultaneously, effectively time-sharing multiprocessors. In GPGPU architectures, fine-grained communication can only occur between small fixed groups of

threads that execute in parallel on a single SM, eliminating the need for co-scheduling in GPGPU architectures.

Massively-parallel SIMD architectures have also been researched outside of the context of GPUs. The MasPar MP-1 architecture [44] and the Connection Machine 1 and 2 [45] are two examples of massively-parallel non-GPU SIMD architectures. They differ from GPU architectures in that they are only capable of executing a single instruction stream on the SIMD array at a time. Modern GPUs contain many stream processors each capable of simultaneously executing an independent instruction stream and switching between a large number of threads to hide main memory latencies. This allows the GPU to achieve much higher thread-level parallelism than the MasPar MP-1 or Connection Machine architectures. Additionally, the Connection Machine processing elements operate on single-bit data while GPU processing elements are typically optimized for 32-bit floating-point data.

Du *et al.* propose a dynamic scheduling mechanism for heterogeneous grid computing environments [46]. Their scheduling mechanism considers the cost of migration when deciding whether to migrate a task to another machine in the grid. Grid computing is quite different than GPGPU processing, and has very different bandwidths and latencies. However, the process of considering the cost of migration when deciding whether to migrate a task is similar to considering the cost of preemption when redistributing resources to processes on the GPU. Thus, similar mechanisms may apply to both.

Fung *et al.* propose scheduling heuristics for dynamically regrouping threads on the GPU in order to reduce overhead from divergent branches [13]. GPGPU performance is generally enhanced when all the threads executing on a single SIMD processor are executing the same instructions. The research of Fung *et al.* aims to dynamically change what threads execute together on a SIMD in order to maximize the SIMD utilization. Currently, GPU threads are statically grouped onto SIMDs, and this grouping does not change even if performance suffers due to diverging control flow, where parallel threads

then each execute different instruction streams. This thread regrouping research is similar to the SM scheduling evaluation in Chapter 6 in that regrouping involves thread scheduling on the GPU. However, the regrouping research assumes all threads are from a single process, while my research aims to schedule threads from multiple processes on the same GPU. The research of Fung *et al.* is complementary to my research and both could be merged on a real system.

A number of works have targeted allowing a single piece of source code to be written for multiple parallel platforms [47, 48, 49, 50, 51, 52, 53]. Having multiple architectures to execute a single piece of code enables greater scheduling flexibility through greater hardware choices on heterogeneous systems and is complementary to the work evaluated in this document. For example, an application kernel written in CUDA can run on several platforms, but only performs well on the GPU. If CUDA code performed almost as well on the CPU as it does on the GPU, then when the GPU was busy but the CPU was idle, the code could be executed on the CPU. MCUDA is a compiler and runtime environment that allows code written in CUDA to also run on shared-memory multicore processors [47]. GPUOcelot emulates NVIDIA's virtual PTX GPU ISA to allow CUDA applications to execute GPGPU kernels on the CPU. That project aims to provide a Just-In-Time compiler and re-translator to allow CUDA applications to run natively on the CPU [48, 49]. Lee *et al.* present a source-to-source translator which converts C source code that utilizes OpenMP to CUDA source code [50] and Papakonstantinou *et al.* present a tool-flow for synthesizing CUDA source code for execution on FPGAs [51]. The RapidMind development platform is a parallel programming interface for the Cell multiprocessor, AMD and NVIDIA GPUs, and x86 multicore processors [52]. RapidMind allows source code to be written once within the RapidMind framework and then compiled to any of the supported target platforms. The *Open Computing Language* (OpenCL) is a cross-platform parallel programming interface that is supported by a number of companies including AMD, Apple, Intel and NVIDIA. OpenCL

is relatively new and should provide a consistent interface for parallel programming on several platforms including GPUs, multicore CPUs, and CPUs with vector instructions [53].

Jiménez *et al.* assume binaries are available to execute on either the CPU or GPU [54]. Using this assumption they develop predictive dynamic scheduling heuristics for executing portions of applications on either the CPU or the GPU. The scheduling is automatic, rather than statically-determined by the programmer. My research does not consider scheduling functions on the CPU, but instead assumes the programmer, operating system, or runtime environment has already assigned multiple processes to the GPU. The work of Jiménez *et al.* could be implemented in combination with spatial multitasking.

Kato *et al.* propose a real-time GPU scheduler for providing QoS to GPU applications [55]. Their scheduler time-multiplexes applications on the GPU by controlling commands issued to the GPU at the device-driver level. Rossbach *et al.* present operating system abstractions for providing priority-based scheduling and performance isolation for GPGPU applications [56]. These scheduling and QoS mechanisms are built on top of existing cooperative multitasking device drivers and GPGPU-capable hardware. Elliott and Anderson evaluate real-time scheduling in a mixed CPU and GPGPU system [57]. Like Rossbach *et al.* and Kato *et al.*, they assume the GPU is a single compute unit and consequently implement their real-time schedulers on top of cooperative multitasking. In contrast, this research examines QoS on top of spatial multitasking instead of temporal multitasking.

In summary, my research evaluates spatial versus cooperative multitasking for general purpose applications on the GPU. NVIDIA's Fermi architecture spatially shares the GPU among kernels from a single application. In contrast to Fermi my research evaluates true multitasking of GPGPU kernels from multiple applications. Time-sharing versus space-sharing of compute resources has been evaluated in the context of multicore CPUs and several GPU-like architectures also support spatial sharing. However compared to GPUs,

the scale of parallelism in these architectures is considerably less and individual cores tend to be more complex.

I also evaluate GPU SM partitioning heuristics and QoS when applications share the GPU via spatial multitasking. A large body of work exists on scheduling compute resources among multiple applications, however existing work does not evaluate dividing GPU SMs among applications spatially sharing the GPU. Some work has been done in developing QoS for GPGPU applications. However, this previous work does not consider changing the multitasking model of the GPU and instead builds QoS on top of existing temporal multitasking mechanisms while my work examines QoS on spatial multitasking.

# 3 METHODOLOGY

This chapter presents an overview of the applications, simulation environment and common methodology used for this research.

## 3.1 Application Descriptions

I characterize workloads targeting the portable and set-top markets. These workloads are selected from source code available online [58]. Although the workloads and input data sets target set-top and portable devices, the results should also apply to desktop devices. I have placed all workloads and input data online for public use [59]. Some workloads include CPU computation as detailed in the following workload descriptions; however, I profile only the GPU portions of computation, as this is the portion directly impacted by spatial multitasking. The workloads are categorized in Table 3.1.

**Ray Tracing [27]** implements a rendering engine that supports shadows and reflections up to five levels deep. The CPU performs object creation and movement. A single GPU kernel performs rendering and is executed each time a new frame is created.

**AES [28]** supports 128- and 256-bit *Advanced Encryption Standard* (AES) encryption and decryption. A single GPU kernel performs the entire encryption/decryption algorithm. The CPU performs initialization, cleanup, and I/O before and after the encryption/decryption. For this study, a 128-bit key is used to encrypt 4MB of randomly generated data.

**RSA [60]** performs RSA asymmetric encryption. As with AES, a single GPU kernel performs the entire encryption. The CPU performs some initialization and cleanup. I use a 1024-bit key to encrypt in parallel 128 randomly generated 128-byte messages, for a total encryption size of 16KB. A small input data set is chosen because RSA is typically used to validate credentials or encrypt a secret key for symmetric-key cryptography.

Table 3.1: Workload types.

| Category | Workload | Kernel/Full Application |
|---|---|---|
| Graphics Processing | Fractals | Full |
| | Ray Tracing | Full |
| Encryption/Hashing | AES | Full |
| | RSA | Full |
| | SHA1 | Full |
| Image Processing | JPEG Encode (DCT/Quantize) | Kernel |
| | JPEG Decode (IDCT) | Kernel |
| | Image Denoising | Kernel |
| Sorting | Radix Sort | Kernel |
| Video Processing | DVC | Full |
| | SAD | Kernel |

**SHA1 [61, 62]**, part of the StoreGPU library, provides an API for SHA1 and MD5 hashing on either an entire block of data or a sliding window of data. A single GPU kernel computes all hashes; the CPU performs initialization, I/O, and cleanup. For this study, I hash 4MB of randomly generated data.

**JPEG Encoding and Decoding [29]** are a partial implementations of JPEG encoding and decoding. I treat encoding and decoding as two separate workloads. Encoding consists of two GPU kernels: one for the *Discrete Cosine Transform* (DCT) and one for quantizing the DCT's output. Decoding consists of a single *Inverse Discrete Cosine Transform* (IDCT) GPU kernel. For both encoding and decoding, multiple threads in the GPU kernels work in parallel on 8x8 blocks of data. For encoding, the DCT input is a 3072x2304-pixel bitmap, and the quantization input is the DCT output. The decoding input is the encoding output, which is pre-generated for this study.

**Fractals [30]** renders a number of fractals overlayed onto a single image. Multiple frames can be rendered to animate the fractals in video. The resulting images and videos are used as artwork and screensavers. This is a GPGPU port of the electric sheep program. The application is made up of a number GPU kernels that perform the rendering in several

Table 3.2: Application kernel configuration broken down by percentage of total execution time and number of times each kernel is executed in a single run of the application.

| Application | Kernel | Exec. Time | Calls | Blocks | Threads per Block | Blocks per SM |
|---|---|---|---|---|---|---|
| AES Decrypt | aesDecrypt128 | 100.0% | 1 | 4,097 | 256 | 5 |
| AES Encrypt | aesEncrypt128 | 100.0% | 1 | 4,097 | 256 | 3 |
| DVC | motion_copy_2ref_2b | 19.2% | 98 | 96 | 256 | 2 |
| | s_transform_h_9_3 | 19.1% | 387 | 240 | 256 | 4 |
| | motion_copy_2ref_4b | 16.6% | 49 | 176 | 256 | 2 |
| | s_transform_v_9_3 | 12.8% | 150 | 5 | 128 | 1 |
| | s_transform_v_9_3 | 7.2% | 150 | 3 | 128 | 1 |
| | s_transform_v_9_3 | 5.0% | 50 | 10 | 128 | 1 |
| | s_transform_v_9_3 | 4.3% | 150 | 2 | 128 | 5 |
| | s_transform_h_9_3 | 3.5% | 150 | 120 | 256 | 4 |
| | add_s16_s16 | 3.2% | 141 | 256 | 256 | 4 |
| | s_transform_h_9_3 | 2.0% | 150 | 60 | 256 | 2 |
| | s_transform_v_9_3 | 2.0% | 100 | 1 | 128 | 1 |
| | upsample_vertical | 1.9% | 15 | 100 | 256 | 3 |
| | upsample_vertical | 1.8% | 30 | 50 | 256 | 2 |
| | s_transform_h_9_3 | 1.5% | 150 | 30 | 256 | 1 |
| Fractals | resetPointsKernal | 99.8% | 7 | 512 | 64 | 8 |
| | setBufferKernal | 0.2% | 2 | 336 | 256 | 4 |
| Image Denoising | NLM2 | 100.0% | 1 | 110,592 | 64 | 8 |
| JPEG Decode | 2IDCT | 100.0% | 1 | 13,824 | 64 | 8 |
| JPEG Encode | QuantizationFloat | 62.9% | 1 | 110,592 | 64 | 8 |
| | 2DCT | 37.1% | 1 | 13,824 | 64 | 7 |
| RSA | kern_RSAenc_Montg | 100.0% | 1 | 4 | 32 | 3 |
| Radix Sort | radixSortBlocksKeysOnly | 51.9% | 8 | 256 | 256 | 3 |
| | findRadixOffsets | 43.6% | 16 | 512 | 256 | 4 |
| | scan4 | 3.0% | 16 | 8 | 128 | 1 |
| | scan4 | 1.5% | 8 | 1 | 128 | 1 |
| Ray Tracing | render | 100.0% | 33 | 2,048 | 128 | 8 |
| SAD | mb_sad_calc | 90.0% | 7 | 4,800 | 61 | 8 |
| | larger_sad_calc_8 | 7.9% | 6 | 300 | 128 | 8 |
| | larger_sad_calc_16 | 2.1% | 6 | 300 | 32 | 8 |
| SHA1 | sha1_kernel_direct | 100.0% | 1 | 65 | 64 | 3 |

steps while the CPU is responsible for reading the parameters for generating the fractals and encoding the GPU-computed results as images or videos.

**Image Denoising [63]** removes white noise from an image using a modified version of the Non Local Means filter. A single GPU kernel performs filtering; the CPU performs

initialization, cleanup, and I/O. In practice, image denoising may be applied to still images in conjunction with other filters and edits, or may be applied to a stream of video frames. The input data is a 3072x2304-pixel bitmap.

**Radix Sort [64, 65]** is a single GPU kernel from the *CUDA Data Parallel Primitives* (CUDPP) library. It performs a radix sort of an array of $2^{18}$ integers that are randomly generated on the CPU.

**DVC [31]** is an implementation of the decoding portion of the *Dirac Video Codec*, a modern video codec intended for streaming high-definition video. A notable difference between Dirac and H.264 is that Dirac uses wavelet transforms rather than cosine transforms. DVC implements the 2D inverse wavelet transform, motion compensation decoding, and up-sampling on the GPU. All other decoding functions take place on the CPU. For this study, the input video is 39 frames at 320x240 resolution.

**SAD [66]** is part of the Parboil benchmark suite for GPUs. It implements the *Sum of Absolute Differences* (SAD), which is used by a number of video encoders, including H.264. SAD is made up of three GPU kernels. The first computes the SAD for 4x4-pixel blocks. The second uses the results of the first to compute the SAD for 8x8-pixel blocks. The final kernel uses the output of the second kernel to compute the SAD for 16x16-pixel blocks. The CPU is responsible for setup, cleanup, and I/O. For this study, the SAD is computed for two 320x240-pixel frames.

The amount of parallelism available in each application plays an important role in how efficiently GPU resources are used. Details of GPU kernel configurations are presented in Table 3.2. For each application I present the percentage of total GPU time spent executing each kernel, the total number of calls made to each kernel for a single run of the application, and number of thread blocks and threads per block in each kernel.

In CUDA, a block cannot be split across multiple SMs, so applications with fewer blocks are less likely to fully utilize available SMs. For example, RSA has 4 blocks and 32 threads

Table 3.3: Baseline GPGPU-Sim configuration, NVIDIA Quadro FX 5800.

| | | | |
|---|---|---|---|
| SMs | 30 | SM Freq. | 650MHz |
| Memory Controllers | 8 | Memory Freq. | 800MHz |
| Interconnect Model | Crossbar | Interconnect Freq. | 650MHz |
| Registers | 64KB/SM | Warp Size | 32 Threads |
| Texture Cache | 8KB/SM | Constant Cache | 8KB/SM |
| Shared Memory | 16KB/SM | | |

per block, this means RSA never utilizes more than 4 SMs. 32 threads per block is also relatively small, so RSA is unlikely to efficiently use even 4 SMs. Kernels with a small number of threads per block require more blocks per SM in order to fully utilize the SM.

## 3.2 Simulation Environment

I simulate CUDA applications with GPGPU-Sim, a cycle-accurate execution-driven GPU simulator capable of simulating unmodified CUDA and OpenCL applications (but not graphics applications) [14, 67]. I model an NVIDIA Quadro FX 5800 GPU, which implements the GT200 architecture described in Section 2.2. GPGPU-Sim has been used in a number of publications and has been shown to correlate well with existing hardware [14].

GPGPU-Sim emulates NVIDIA's virtual GPU instruction set architecture called *Parallel Thread eXecution* (PTX). GPU portions of both CUDA and OpenCL applications are emulated on the CPU using a cycle-accurate GPU performance model. CPU portions of benchmarks are run directly on native hardware to provide functional correctness. The simulator does not, however, provide a performance model for the CPU portions of benchmarks or model the overhead of data transfers between the CPU and GPU. I use the default GPGPU-Sim parameters that approximate the NVIDIA Quadro FX 5800 GPU [67, 68], shown in Table 3.3. Although this infrastructure models an NVIDIA GPU, these research results should apply to other GPU architectures.

NVIDIA's CUDA tool-chain compiles programs into NVIDIA's PTX virtual ISA. When the PTX code is run in real hardware, it is Just-In-Time compiled to the ISA implemented by the GPU. This unpublished hardware ISA changes with each generation of GPUs. The PTX virtual ISA provides a stable interface across multiple generations of GPUs. GPU-Sim emulates the PTX virtual ISA and provides custom CUDA runtime libraries. This enables CUDA programs to be compiled for the simulator without source code modification. However, some work was still required to simulate the applications presented in Section 3.1. GPGPU-Sim did not implement all of the CUDA API required for these applications so GPGPU-Sim had to be extended.

Internally, GPGPU-Sim uses a modified version of SimpleScalar for performance modeling and a custom implementation of the CUDA runtime libraries to functionally simulate CUDA applications. The GPGPU-Sim custom CUDA runtime libraries are dynamically linked with the application that is being simulated. The runtime libraries functionally emulate the CUDA GPU API on the CPU and drive the GPU performance model. The GPU performance model consists of a cycle-accurate custom GPU micro-architectural model combined with the BookSim interconnect simulator.

The CPU portions of benchmarks are run directly on native hardware to provide functional correctness in the simulations. The simulator does not currently provide a performance model for the CPU portions of benchmarks or model the overhead of data transfer between the CPU and GPU. For this research I aimed to evaluate the performance potential of spatial multitasking compared to cooperative multitasking when multiple applications share the GPU. I did not evaluate the frequency and duration of GPU multitasking and how CPU computation and communication overhead affects this, due to the unpredictable dependence on future devices, workloads, and users. Consequently, CPU and PCI-Express performance modeling was deemed unnecessary for the methodology used in this research.

### 3.2.1 GPGPU-Sim Multitasking Modifications

For this research I modified GPGPU-Sim to support spatial and temporal multitasking, for clarity I will call this GPGPU-Sim-multitasking. Originally GPGPU-Sim only supported simulating a single GPU application. GPGPU-Sim is able to simulate unmodified CUDA binaries by replacing NVIDIA's dynamically-linked libcuda library with a custom version of libcuda that contains GPGPU-Sim. CPU portions of the application being simulated run natively on the host CPU, but when calls are made into the CUDA library, GPGPU-Sim emulates them instead of the CUDA runtime. On the first call to libcuda, GPGPU-Sim extracts the GPU kernels, encoded in the PTX virtual GPU ISA, from the binary using the CUDA compiler. This PTX code is then simulated when the application launches a GPU kernel. This means communication between the application and the simulator takes place through function calls in libcuda.

GPGPU-Sim-multitasking changes GPGPU-Sim to be a stand-alone executable. This stand-alone executable forks applications that are to be simulated into their own process. Like GPGPU-Sim, GPGPU-Sim-multitasking replaces NVIDIA's dynamically-linked libcuda library with a custom version. However, the GPGPU-Sim-multitasking version of libcuda differs from the original GPGPU-Sim version. GPGPU-Sim-multitasking libcuda uses named pipes to communicate between the application and the simulator, instead of function calls and the the simulation model has been pushed into the stand-alone GPGPU-Sim binary. By running applications in a separate process from the simulator (and other applications) and by replacing libcuda GPGPU-Sim-multitasking is able to simulate unmodified CUDA binaries. Because communication and synchronization occurs between applications and GPGPU-Sim through the CUDA API, the overhead of a simple but slow inter-process communication like named pipes is acceptable. Differences between CUDA, GPGPU-Sim, and GPGPU-Sim multitasking are highlighted in Figure 3.1.

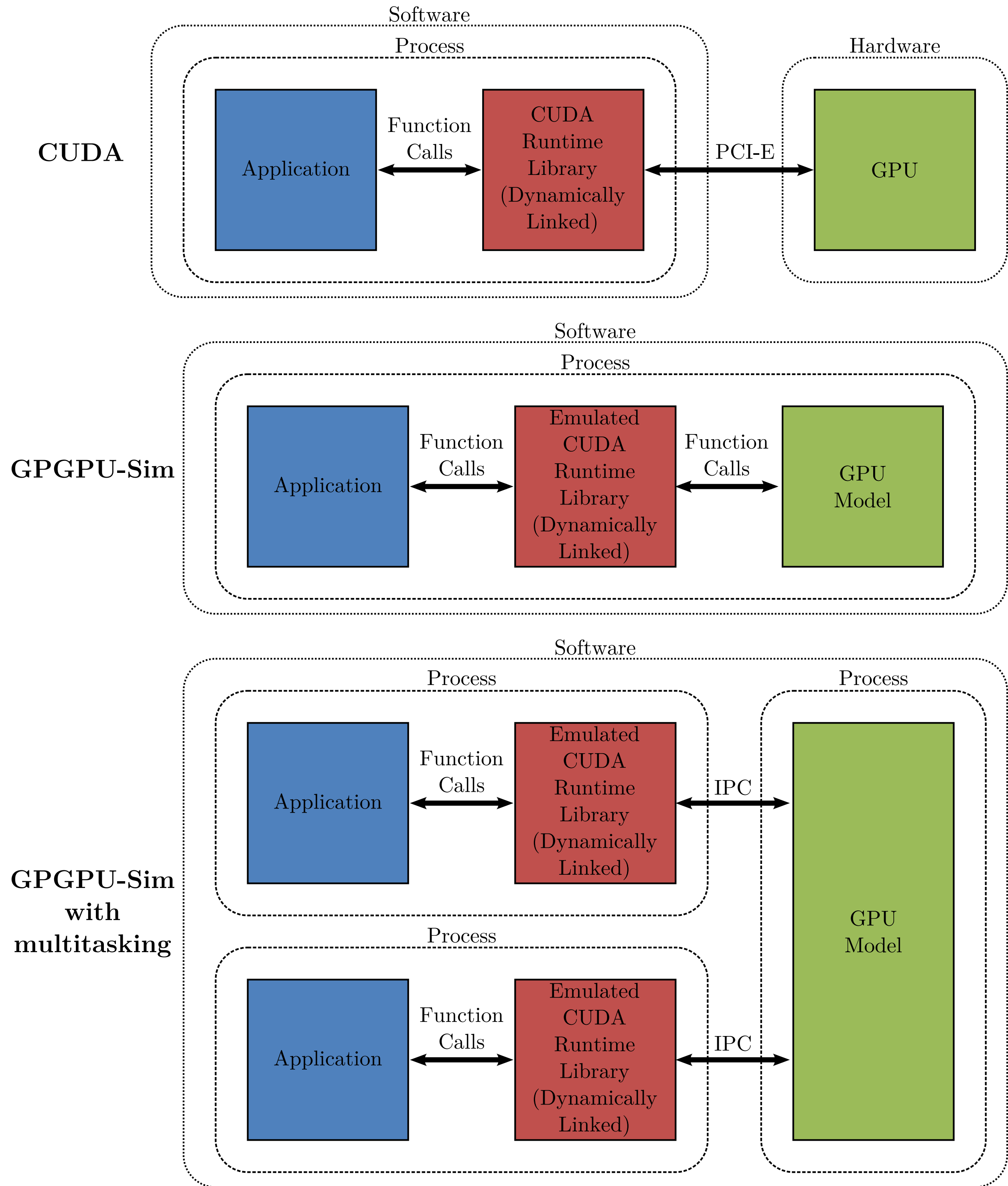



Figure 3.1: Comparison of CUDA, GPGPU-Sim and GPGPU-Sim-multitasking.

```
# Statically partition 12 SMs for Radix_Sort
# and 18 SMs for Ray_Tracing.
SCHEDULER = static_split
SCHEDULER_OPTIONS = 12 18

# Restart any jobs that complete early.
RESPAWN_PROCESSES = TRUE

# Applications to run.
0 0 ./Radix_Sort --n=262144 --iterations=1 --keysonly --quiet
0 0 ./Ray_Tracing --frames=1000
```

Figure 3.2: Example GPGPU-Sim-multitasking configuration file.

In addition to the multiprocess changes to GPGPU-Sim I also had to introduce extra multitasking related configuration and SM partitioning to GPGPU-Sim multitasking. I have added a multitasking configuration file that specifies to the simulator what SM partitioning heuristic to use, what applications to simulate and when, in simulated time to launch these applications. A typical multitasking configuration file is shown in Figure 3.2. In this example Radix Sort and Ray Tracing are simulated. Both applications start at the same time and will be restarted if they complete execution before the simulation is terminated (simulation length is specified in the standard GPGPU-Sim configuration file). The applications share the GPU via spatial multitasking. Radix Sort will execute on 12 SMs while Ray Tracing will be assigned 18 SMs. Figure 3.3 fully specifies the format of the GPGPU-Sim-multitasking configuration file.

I have added an interface for schedulers that choose how SMs are partitioned among applications. This interface is specified in Figure 3.4. A scheduler must inherit the `scheduler` class and the `scheduler::create` function must be extended to construct the new type of scheduler. A scheduler must implement `scheduler::schedule`. This function takes as arguments a list of processes that are currently using the GPU, a vector that maps processes onto SMs, and the current simulation time in SM cycles. A scheduler must assign pro-

```
# Lines beginning with the # character are comments.
SCHEDULER = The selected multitasking scheduler.
SCHEDULER_OPTIONS = A configuration string passed to the scheduler, format
                    is scheduler specific.
RESPAWN_PROCESSES = Restarts jobs that complete before simulation ends.
TERMINATE_AFTER_FIRST_PROCESS_COMPLETES = Ends simulation as soon as any
                                          application completes execution.
# Applications to simulate are specified as follows:
[Start time] [SM cycles to simulate the application for] [binary] [args]
```

Figure 3.3: General GPGPU-Sim-multitasking configuration file format.

cesses to SMs. This mapping is returned in the `SM_to_process_map` vector. The scheduler must also return the next time, in SM cycles, that `scheduler::schedule` should be called. GPGPU-Sim-multitasking will call `scheduler::schedule` in the following situations:

- A new GPU kernel is launched.
- A GPU kernel has completed execution.
- The time returned by the last call to `scheduler::schedule` has been reached.

GPGPU-Sim-multitasking currently does not support the ability to re-assign an SM from one running GPGPU kernel to another. This limits the functionality of schedulers currently implemented in GPGPU-Sim-multitasking. For this research I have implemented a cooperative multitasking scheduler and a static-split spatial multitasking scheduler. The static-split scheduler partitions SMs among applications as specified in the multitasking configuration file, as shown in Figure 3.2. This partitioning remains static for the duration of the simulation. This allowed me to examine complex, but static, partitioning heuristics that can form the building blocks for dynamic SM partitioning. For this research, these partitioning heuristics are implemented in python and generate the appropriate multitasking configuration file after determining what the static SM partitioning should be for the specified applications sharing the GPU. I have also implemented several dynamic

```
#define NO_RESCHEDULE numeric_limits<uint64_t>::max()

class scheduler {
 public:
  static scheduler *create(string scheduler_name, string scheduler_options);

  virtual uint64_t schedule(list<process *> &running_processes,
                            vector<process *> &SM_to_process_map,
                            uint64_t current_simulation_time);
};

scheduler *scheduler::create(string scheduler_name,
                             string scheduler_options) {
  if (strcasecmp(scheduler_name.c_str(), "cooperative") == 0) {
    return (scheduler *) new scheduler_cooperative(scheduler_options);
  } else if (strcasecmp(scheduler_name.c_str(), "preemptive") == 0) {
    return (scheduler *) new scheduler_preemptive(scheduler_options);
  } else if (strcasecmp(scheduler_name.c_str(), "static_split") == 0) {
    return (scheduler *) new scheduler_static_split(scheduler_options);
  }
  cout << "Error unknown scheduler:  " << scheduler_name << endl;
  exit(-1);
}
```

Figure 3.4: GPGPU-Sim-multitasking scheduler interface.

partitioning schedulers as examples for future GPGPU-Sim-multitasking use, including a pre-emptive multitasking scheduler. However, at this time these schedulers are untested.

## 3.3 Simulation Length

I chose to limit GPGPU-Sim spatial multitasking simulations to 5M GPU cycles in order to place reasonable constraints on real world simulation time, this corresponds to 7.7ms of simulated time. Figure 3.5 plots the speedup of spatial multitasking compared to cooperative multitasking versus simulated GPU cycles. The details of spatial multitasking and an analysis of the performance of spatial multitasking are discussed further in Chapter 5.

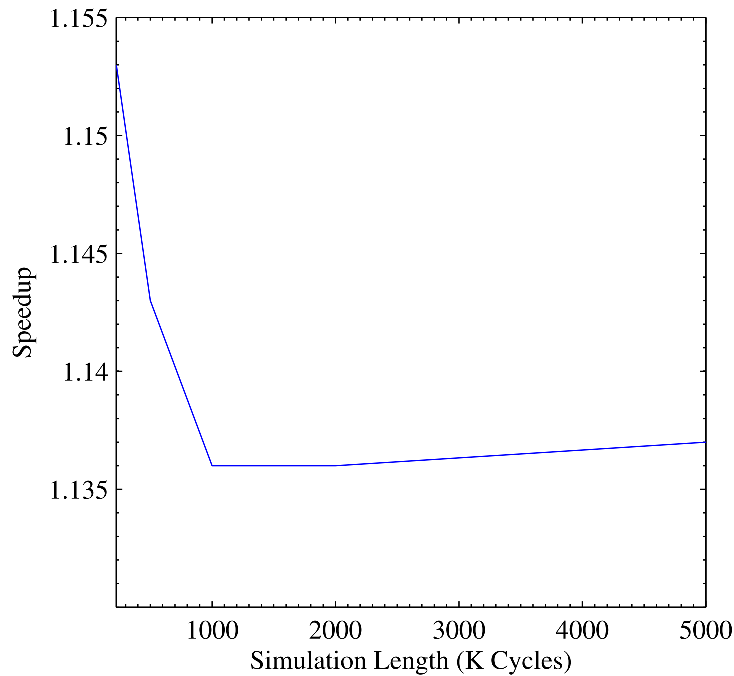



Figure 3.5: Speedup of spatial multitasking compared to cooperative multitasking versus simulation length. The geometric mean speedup of all combinations of two applications sharing the GPU is presented.

Here this data is briefly presented this data to justify the decision to limit simulations to 5M GPU cycles. As shown in Figure 3.5 there is a significant variation in observed performance of spatial multitasking between 225K cycles and 1M cycles. However, from 1M to 5M cycles the performance variation is much smaller. This is likely because startup variations and application phase behavior are significant compared to the number of cycles simulated, however simulations longer than 1M cycles average this behavior out, resulting in little change in measured performance. I felt simulating 5M GPU cycles was sufficiently long to overcome these variations, while still being reasonably short in terms of real world time.

## 4.1 Methodology

As an initial step in evaluating spatial multitasking, I profiled the applications from Section 3.1 to see how they responded to different amounts of available GPU resources. Specifically, I executed each application in isolation for 5M GPU cycles, and measured the number of instructions simulated while varying the amount of memory and interconnect bandwidth and the number of SMs. The NVIDIA Quadro FX 5800 baseline values for these parameters are shown in Table 3.3. Varying these parameters gives some insight into predicted application performance when sharing the GPU via spatial multitasking. However, simulating applications in isolation does not model several factors that could also affect multitasking performance. For example, an application's demand for memory bandwidth can vary dynamically during execution, affecting the remaining bandwidth available to other applications. Also, interconnect contention among applications is not modeled when examining applications in isolation. For these reasons, I implement spatial and cooperative multitasking in GPGPU-Sim and evaluate them through simulation in Chapter 5.

## 4.2 Evaluation

Figures 4.1 to 4.3 show how the performance of the GPU portion of each application is affected by changes to a specific GPU configuration parameter when applications are executed in isolation on the GPU. Sub-linear speedup (below the dashed line) indicates the application is underutilizing the examined resource either due to a bottleneck in the system or lack of parallelism in the application. In each figure, the parameters that are not varied are set according to Table 3.3.

In Figure 4.1, the number of SMs in the simulated GPU is varied. Speedup is normalized to a GPU with a single SM. As expected, many applications fail to scale linearly as the number of SMs are increased. Only AES Decrypt, AES Encrypt, and Image Denoising maintained near-linear speedup up to the maximum number of SMs simulated. Ray Tracing also scaled well, but tapered off for large numbers of SMs. Fractals and JPEG Decode scaled well to the baseline configuration (30 SMs) but performance leveled off shortly beyond that. Most of the other applications scaled well initially, but leveled off as the number of SMs increased. Most importantly, Figure 4.1 shows that many applications scale sub-linearly up to 30 SMs (the number in the baseline configuration), and scale even worse at expected future resource levels. As expected from the thread configuration shown in Table 3.2, RSA showed almost no performance change as SMs were increased.

Figure 4.2 examines global memory frequency to study the effect of memory bandwidth limits on GPGPU applications; I use memory frequency as an approximation for bandwidth because fixed-frequency cannot easily be modeled. Speedup is normalized to a 200MHz GPU memory clock. All the applications studied showed sub-linear speedup before reaching even the baseline memory frequency (800MHz), indicating these GPGPU applications underutilized memory bandwidth. JPEG Decode, JPEG Encode, SAD, and SHA1 showed the highest demand for memory bandwidth, while AES Decrypt, AES Encrypt, Fractals, Image Denoising, and RSA showed very little demand.

In Figure 4.3, the GPU interconnect frequency is varied. Speedup is normalized to a 150MHz GPU interconnect clock. JPEG Encode, SAD, and SHA1 showed near-linear speedup at higher interconnect frequencies than the other applications, but as shown with memory frequency all of the applications showed sub-linear speedup before the baseline interconnect frequency (650MHz). In this experiment, AES Decrypt, AES Encrypt, Fractals, Image Denoising, and RSA showed little change in performance as interconnect bandwidth increased demonstrating their interconnection bandwidth needs are modest.

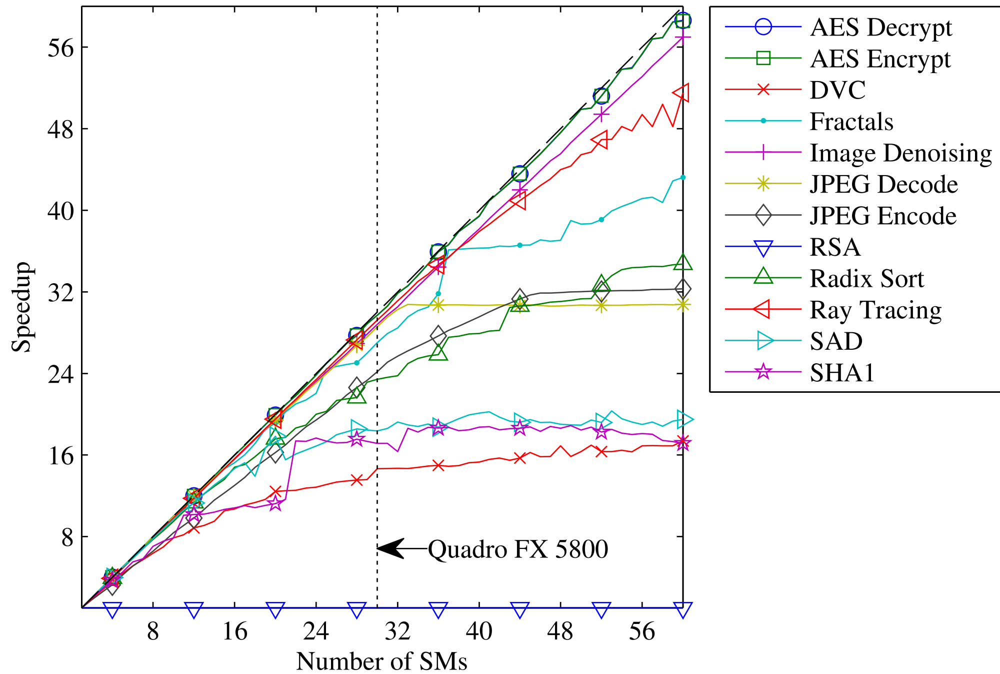



Figure 4.1: Speedup of GPU computation versus number of SMs. Results are normalized to one SM. The dashed line represents linear speedup. The GPU configuration (except SM count) is specified in Table 3.3. Step size is 1 SM.

Figures 4.1 to 4.3 demonstrate that many GPGPU applications underutilize GPU resources. This led to my hypothesis that spatial multitasking will improve system performance by allowing multiple GPGPU applications to share the GPU without lowering their performances significantly. RSA, for example, benefits little from increases in memory bandwidth, interconnect bandwidth, or SMs for the size of the input data simulated in this study. RSA would likely perform just as well using a single SM as if it were assigned the entire GPU. With spatial multitasking, assigning RSA only one SM would still leave the remaining SMs free for other applications.

I have classified each application based on the information provided in Table 3.2 and Figures 4.1 to 4.3.

**Compute bound** applications scale well as the number of SMs are increased but exhibit sub-linear speedup as memory and interconnect frequency is increased. Assuming spatial

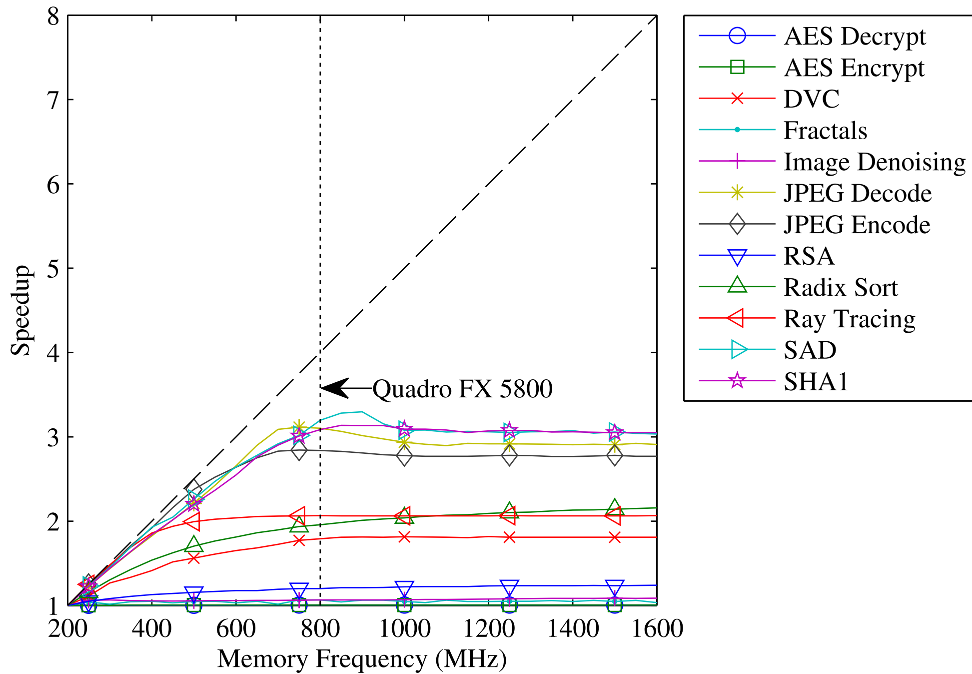



Figure 4.2: Speedup of GPU computation versus GPU memory frequency. Results are normalized to a 200MHz GPU memory clock. The dashed line represents linear speedup. The GPU configuration (except memory frequency) is specified in Table 3.3. Step size is 50MHz.

multitasking is implemented such that SMs are partitioned between applications sharing the GPU, then these applications are unlikely to contribute significantly to increases in total system performance using spatial multitasking. These applications are likely to take twice as long to execute if they are given half the SMs. AES Decrypt, AES Encrypt, Fractals, Image Denoising, JPEG Decode, and Ray Tracing are compute bound.

**Interconnect/Memory bound** applications do not scale well as the number of SMs are increased even though they are written with a sufficient number of threads to potentially make full use of the available SMs. These applications are likely to contribute to an increase in total system performance using spatial multitasking provided their limiting resources do not conflict with other applications sharing the GPU. For example, two memory bound applications sharing the GPU are not likely to improve performance but a memory

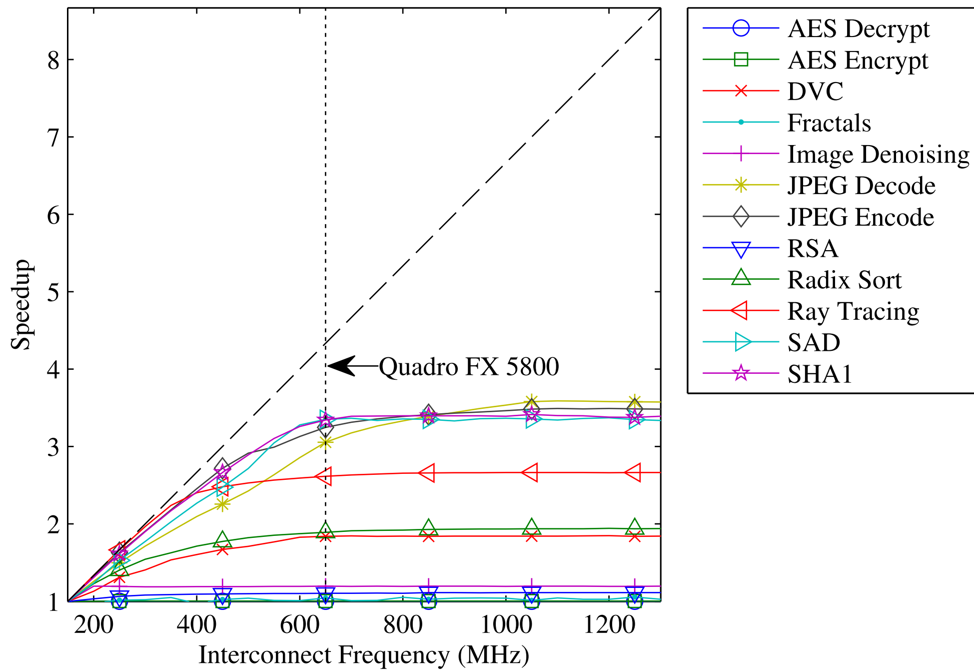



Figure 4.3: Speedup of GPU computation versus GPU interconnect frequency. Results are normalized to a 150MHz GPU interconnect clock. The dashed line represents linear speedup. The GPU configuration (except interconnect frequency) is specified in Table 3.3. Step size is 50MHz.

bound application and compute bound application sharing the GPU is likely to show a performance improvement. JPEG Encode and SAD are interconnect/memory bound.

**Problem-size bound** applications are not written with a sufficient amount of parallelism or lack sufficient input data size to take advantage of all the SMs in the system. These applications are well suited to an implementation of spatial multitasking where SMs are partitioned between applications sharing the GPU. DVC, Radix Sort, RSA, and SHA1 are problem-size bound.

## 4.3 Conclusion

I presented application characterizations that indicate many GPGPU applications fail to utilize GPU resources fully. As future GPUs grow in scale, GPGPU applications are even more likely to underutilize resources. The applications I have characterized show sub-linear speedup as the number of SMs in the GPU are increased and memory and interconnect bandwidth is increased. This indicates, as I show in Chapter 5, that sharing these resources by running multiple GPGPU applications in parallel through spatial multitasking will lead to more efficient resource usage and an improvement in total system performance.

# 5 SPATIAL MULTITASKING

In Chapter 4, I analyzed the performance of GPGPU applications in isolation as the resources available to them are varied. This data indicates that many applications underutilize available GPU resources not only for current hardware but likely even more for future hardware, making spatial multitasking an attractive solution. In this chapter, using simulation, I confirm spatial multitasking is able to out-perform cooperative multitasking when simultaneously executing applications that under-utilize the GPU.

## 5.1 Methodology

I have implemented both cooperative and spatial multitasking in GPGPU-Sim. Using the NVIDIA Quadro FX 5800 baseline configuration, I compare the total run time of two, three and four applications sharing the GPU under spatial and cooperative multitasking. I simulate all combinations, with repetition, of the applications presented in Section 3.1. For example, if there were three applications A, B, and C, then the two-application combinations would be AA, AB, AC, BB, BC, and CC. For this evaluation kernel initialization and cleanup is not modeled for either multitasking method. This favors neither cooperative nor spatial multitasking as they both require this overhead. For my evaluation of spatial multitasking in this chapter, SMs are naively divided evenly among each application. In Chapter 6, I evaluate more intelligent methods of resource partitioning. GPU memory is not virtualized in my implementation of spatial multitasking in GPGPU-Sim. Applications allocate GPU memory from the same physical pool but memory isolation is not enforced. In an implementation of spatial multitasking outside of simulation, memory isolation should be provided to ensure faulty or malicious applications do not read or write another application's memory space.

As discussed in Section 3.3, for many applications it is not feasible to simulate the entire execution in a reasonable amount of time. Instead, I simulate spatial multitasking for a fixed amount of time and measure the work completed (instructions executed per application). When I began experimentation, I tried a number of simulation lengths and compared how the results changed as the length of simulation changed. I found that results from shorter simulations (1M cycles or less) varied considerably, but longer simulations (greater than 1M cycles) converged on similar results. This is probably due to startup variations that are not representative of the steady-state behavior of applications. I simulate for 5M GPU cycles because it is sufficiently greater than 1M cycles that startup variations have minimal impact on performance and is sufficiently small that simulation time is still reasonable for the hardware I had available.

I simulate each combination of applications in spatial multitasking for 5M GPU cycles, if an application completes in less than 5M cycles it is restarted. This ensures the steady-state performance of all applications running is evaluated. I simulate each combination of applications in cooperative multitasking for a fixed number of instructions. The number of instructions each application is run for in cooperative multitasking corresponds to the number of instructions that are completed by the application in the spatial multitasking simulation. This ensures comparison of the same amount of work for each application between spatial and cooperative multitasking [69]. Although I do not model preemptive multitasking for this research, with my chosen methodology preemptive multitasking will always perform worse than cooperative multitasking due to context switch overhead.

## 5.2 Evaluation

Figure 5.1 plots the distribution of speedup of spatial multitasking relative to cooperative multitasking for all combinations of two, three, and four applications sharing the GPU

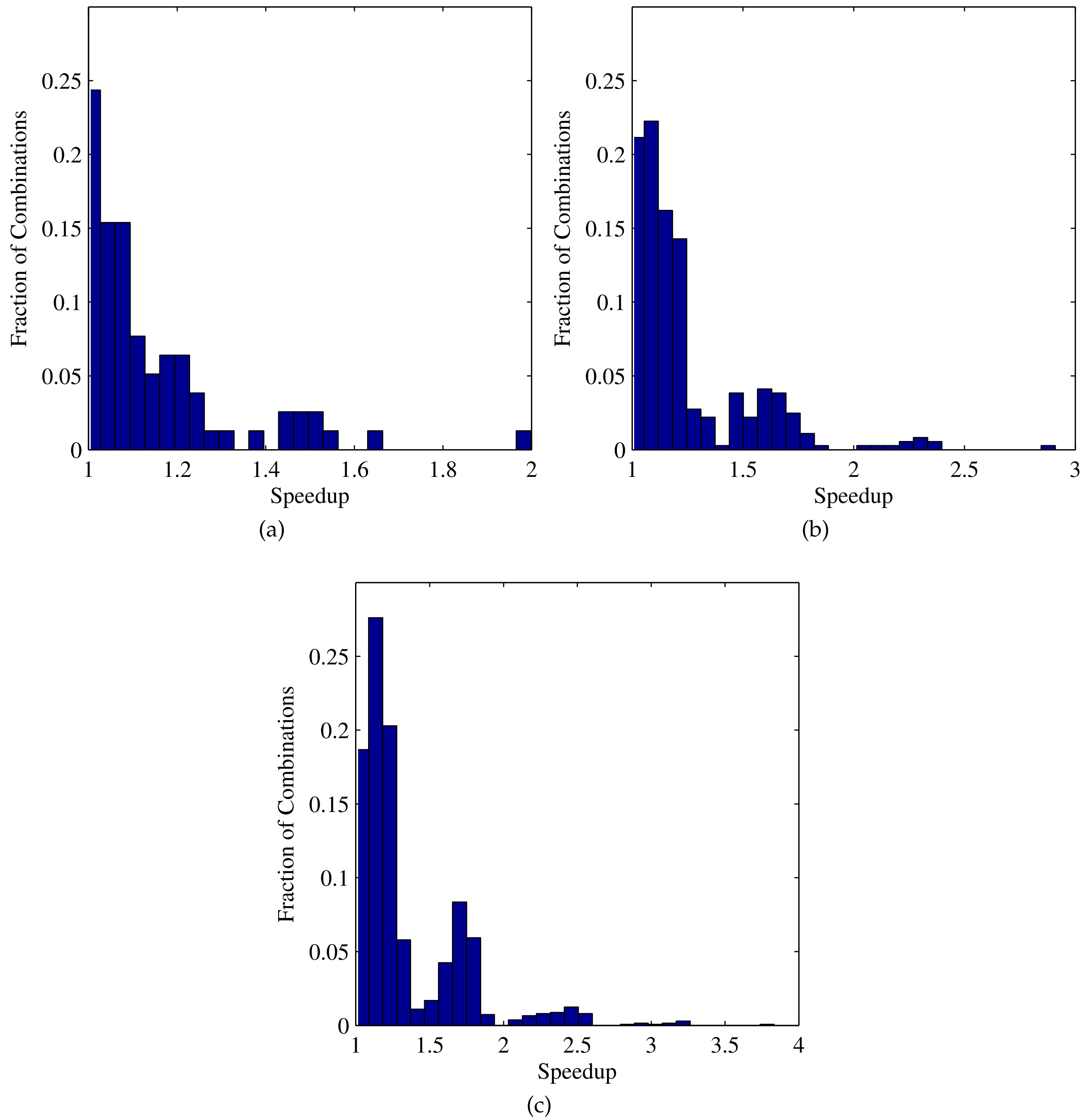



Figure 5.1: Distribution of speedup of spatial multitasking compared to cooperative multitasking. (a) Two applications sharing the GPU. (b) Three applications sharing the GPU. (c) Four applications sharing the GPU.

Table 5.1: Speedup of spatial multitasking compared to cooperative multitasking using a simple even split of SMs among applications.

| | **2 Apps** | **3 Apps** | **4 Apps** |
|---|---|---|---|
| Mean | 1.14 | 1.22 | 1.30 |
| Maximum | 2.00 | 2.91 | 3.83 |
| Minimum | 0.99 | 0.99 | 0.99 |

Table 5.2: Speedup of spatial multitasking compared to cooperative multitasking for several benchmark combinations.

| **Application** | **Classification** | **Speedup** |
|---|---|---|
| DVC | Problem size-bound | 1.14 |
| SHA1 | Problem size-bound | |
| AES Encrypt | Compute-bound | 1.01 |
| Image Denoising | Compute-bound | |
| JPEG Decode | Compute-bound | 1.06 |
| Radix Sort | Problem size-bound | |
| JPEG Encode | Interconnect/memory-bound | 1.05 |
| SAD | Interconnect/memory-bound | |

when SMs are split evenly among the applications. For the applications I have evaluated, in 75% of simulations spatial multitasking shows speedups greater than 1.03, 1.07, and 1.12 for two, three, and four applications, respectively; greater than 1.09, 1.14, and 1.19 in 50% of simulations; and greater than 1.20, 1.26, and 1.55 in 25% of simulations.

Table 5.1 summarizes the results shown in Figure 5.1. Spatial multitasking performs quite well in general and in the worst case only performs slightly worse than cooperative multitasking. As the number of applications sharing the GPU grows, the speedup of spatial multitasking compared to cooperative multitasking also grows due to more efficient utilization of GPU resources.

Table 5.2 details the performance of several specific combinations of applications. DVC and SHA1, two problem-size bound applications, show significant speedup using spatial multitasking compared to cooperative multitasking. JPEG Decode and Radix Sort also show some speedup with spatial multitasking. JPEG Decode is compute bound and

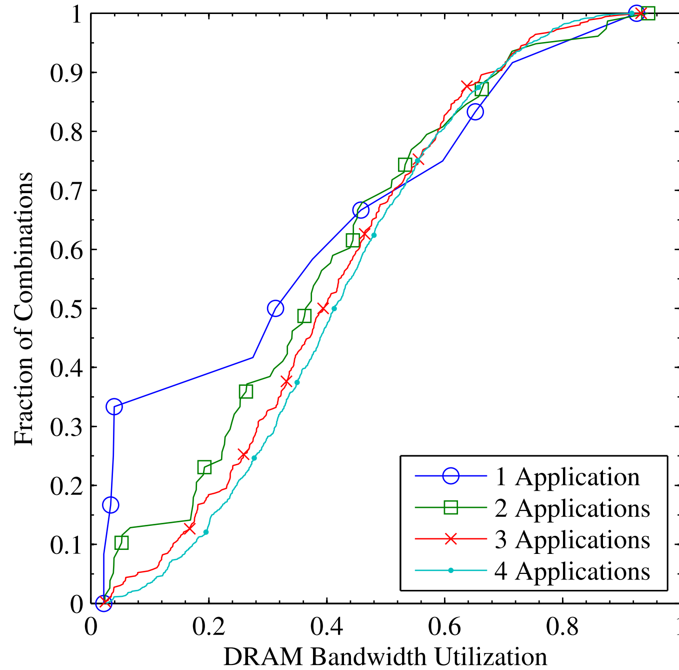



Figure 5.2: Cumulative distribution function of the DRAM bandwidth utilization for all combinations of applications using spatial multitasking. The Y-axis represents the fraction of combinations that have a DRAM bandwidth utilization *less than or equal to* the corresponding point on the X-axis.

Table 5.3: Summary of DRAM bandwidth utilization for all combinations of applications.

| | **1 App** | **2 Apps** | **3 Apps** | **4 Apps** |
|---|---|---|---|---|
| Mean | 37.0% | 38.6% | 40.5% | 42.2% |
| Maximum | 92.6% | 94.5% | 93.3% | 91.7% |
| Minimum | 2.2% | 2.3% | 2.4% | 2.6% |

consequently not well suited for spatial multitasking. However, when JPEG Decode is paired with Radix Sort, which is problem-size bound, spatial multitasking outperforms cooperative multitasking. AES Encrypt and Image Denoising are both compute bound applications and consequently do not show significant speedup using spatial multitasking; however performance is not any worse than cooperative multitasking. Finally, this shows that pairing interconnect/memory bound applications using spatial multitasking, in this case JPEG Encode and SAD, also out-performs cooperative multitasking.

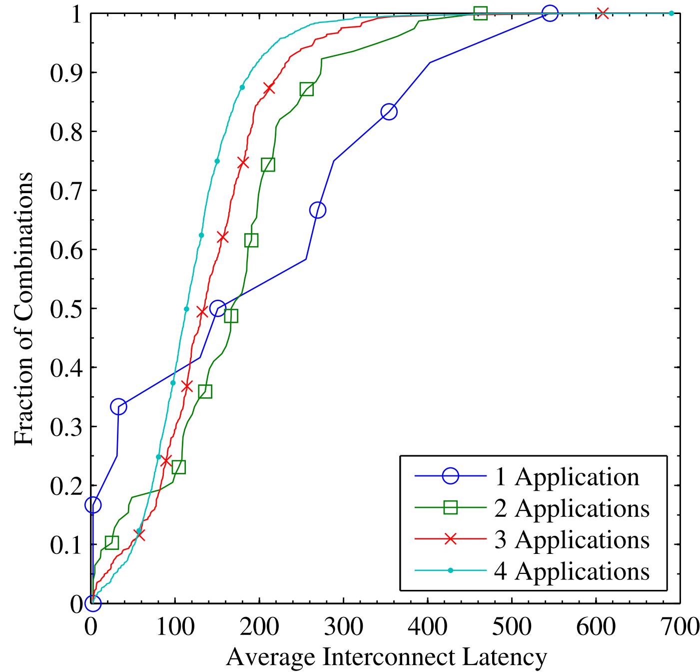



Figure 5.3: Cumulative distribution function of the average interconnect latency, in cycles, for all combinations of applications using spatial multitasking. The Y-axis represents the fraction of combinations that have an average interconnect latency *less than or equal to* the corresponding point on the X-axis.

Table 5.4: Summary of average interconnect latency in cycles for all combinations of applications.

| | **1 App** | **2 Apps** | **3 Apps** | **4 Apps** |
|---|---|---|---|---|
| Mean | 205.5 | 164.5 | 138.8 | 119.0 |
| Maximum | 545.5 | 463.0 | 608.3 | 689.8 |
| Minimum | 2.6 | 2.5 | 2.5 | 2.5 |

Figures 5.2 and 5.3 depict the cumulative distribution function of DRAM bandwidth utilization and average interconnect latency for one to four applications using the GPU under spatial multitasking. Note that one application using spatial multitasking is the same as one application using cooperative multitasking. Tables 5.3 and 5.4 summarize the results shown in Figures 5.2 and 5.3, respectively. These figures and tables show the trend that as the number of applications sharing the GPU under spatial multitasking increases

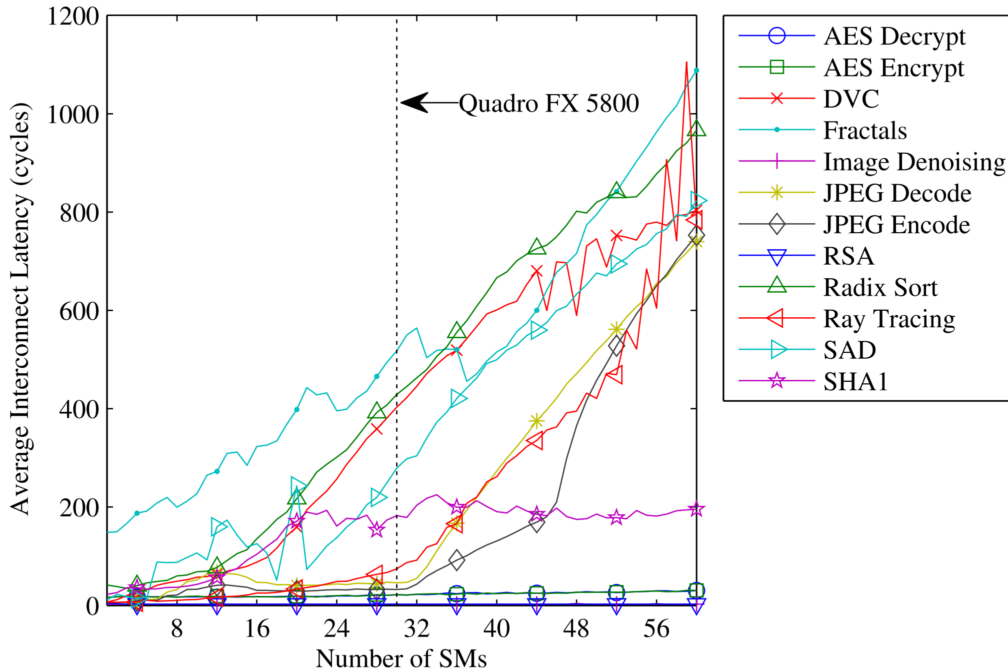



Figure 5.4: Average GPU interconnect latency versus number of SMs. Step size is 1 SM.

the required DRAM bandwidth also increases. However, the average interconnect latency decreases as the number of applications sharing the GPU increases.

The reduction in average interconnect latency using spatial multitasking compared to cooperative multitasking is the result of bursty interconnect traffic. When a single application executes in isolation on the GPU all 30 SMs often simultaneously contribute to bursts of interconnect traffic. However in spatial multitasking, each application executes on a subset of SMs. Interconnect traffic bursts from different applications are not aligned, therefore most of the time only a subset of SMs contribute to a burst in interconnect traffic which decreases the average interconnect latency. Figure 5.4 shows the average interconnect latency of each application as the number of SMs available to the application is varied. Each application is executed in isolation. This shows for many applications a 2x reduction in SMs results in more than a 2x reduction in average interconnect latency. This contributes to a reduction in average interconnect latency for spatial multitasking.

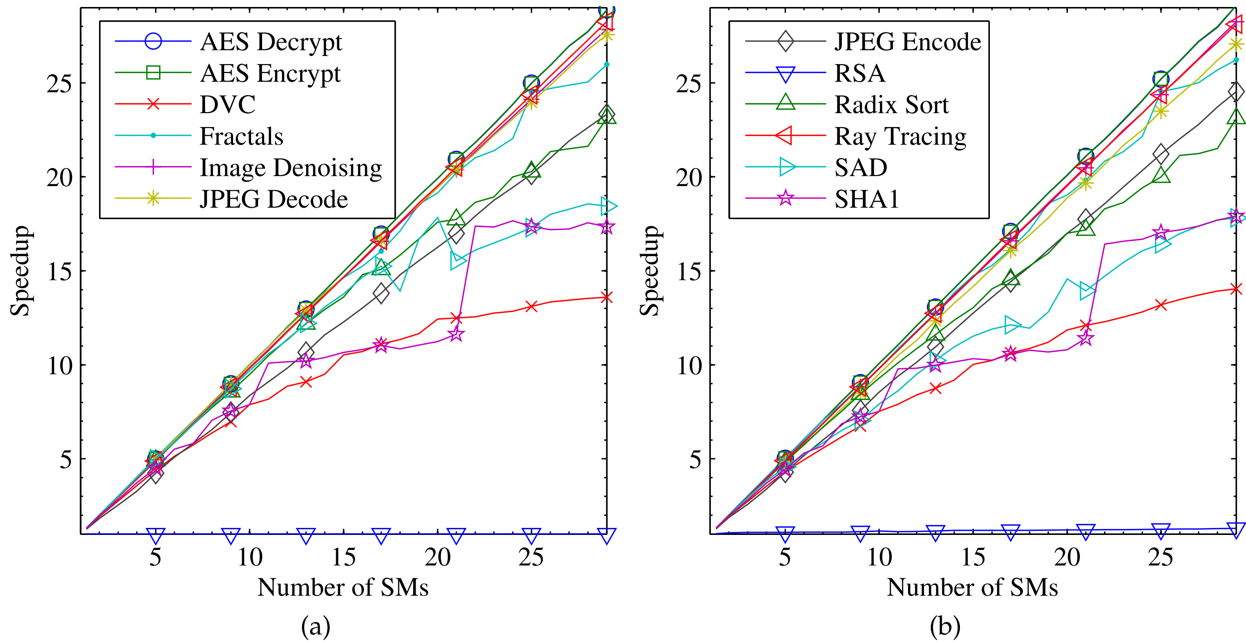



Figure 5.5: Performance scaling versus number of SMs an application executes on when running in (a) isolation and (b) spatial multitasking.

Figure 5.5 compares application performance scaling as the number of SMs the application executes on is varied in isolation versus spatial multitasking. Figure 5.5a reproduces a subset of the performance scaling in isolation plot from section 4.2. Figure 5.5b plots the average performance of each application when sharing the GPU with other applications via spatial multitasking. Performance versus the number of SMs an application executes on is very similar in isolation and with spatial multitasking. This makes an individual application's performance in isolation a good predictor of the application's performance when sharing the GPU with other application's via spatial multitasking. To calculate the average performance of an application sharing the GPU in spatial multitasking, I look at all combinations of the evaluated applications sharing the GPU and all possible SM distributions. From this, I calculate the arithmetic mean of total instructions an application completes in 5M simulated GPU cycles. To compute speedup, I normalize average instructions simulated to a single SM.

## 5.3 Implementation Details

In current GPUs, only a single application can execute at a time. Spatial multitasking adds the ability to partition GPU resources among multiple simultaneously-executing applications. However, there are several challenges for both hardware and system software that must be addressed to implement spatial multitasking. In this section, I address these challenges.

In situations where GPU resources are oversubscribed, spatial multitasking needs to also utilize temporal multitasking. For example, if there is not enough GPU memory available for all of the applications sharing the GPU then some applications will have to wait for others to complete or applications can be given time-slices on the GPU. In these situations, a hybrid of temporal and spatial multitasking can be used. For example, if three GPGPU applications are simultaneously executing but there are only enough resources for two applications to share the GPU, then at any time the GPU will be executing two applications and preemption can be used to cycle through the three applications. Thus, spatial multitasking should be implemented in combination with preemptive or cooperative multitasking.

The GPU memory system requires modifications to support multiple applications. Although several GPU architectures already virtualize global and texture memory for a single application [12], the memory subsystem should be enhanced to ensure memory isolation between applications for security, reliability, and programmability purposes. This can be as simple as segmenting memory and assigning each segment to an application or can be a full implementation of virtual memory with protection and paging, similar to modern CPUs. As shown in Table 5.5, memory requirements of set-top and portable applications are likely to be modest, making paging unnecessary. However, for desktop and scientific computing applications memory requirements are likely to be higher.

Table 5.5: Global memory footprint of applications.

| Application | Footprint |
|---|---|
| AES Decrypt | 8.0 MB |
| AES Encrypt | 8.0 MB |
| DVC | 7.1 MB |
| Fractals | 3.5 MB |
| Image Denoising | 54.0 MB |
| JPEG Decode | 27.0 MB |
| JPEG Encode | 27.0 MB |
| Radix Sort | 2.1 MB |
| Ray Tracing | 32.0 MB |
| RSA | 69.5 KB |
| SAD | 26.0 MB |
| SHA1 | 4.0 MB |

The frequent and time-sensitive task of scheduling threads to SMs is currently handled on the GPU without OS interaction. This can remain unchanged for spatial multitasking; however, the scheduler must additionally partition resources among applications. Repartitioning should occur when new applications request to execute on the GPU or an existing application finishes GPU execution. This occurs less frequently than the start and completion of individual threads in an application, and is therefore a more coarse-grained scheduling decision than scheduling individual threads to SMs. Table 5.6 presents a summary of the observed computation times for all the kernels in the applications presented in Section 3.1. Several kernels are executed multiple times. In these cases, I count each execution towards the computation of the mean execution time. The execution times of the kernels observed here are considerably smaller than the typical operating system time slice. For example in the Linux kernel an application with a niceness of 0 (the baseline priority) typically receives a 100ms time slice. However, Table 5.6 does not account for kernel setup and cleanup time because it is not modeled in my simulation environment. This coupled with the fact that each time a GPGPU kernel is spawned or completed GPU driver interaction is required makes the OS or the GPGPU runtime environment the appropriate

Table 5.6: Observed GPGPU kernel runtimes, neglecting setup and cleanup time. Note: RSA did not complete simulation due to time constraints but the kernel was still running after 638ms.

| Application | Kernels Executed | Mean (μs) | Min (μs) | Max (μs) |
|---|---|---|---|---|
| AES Decrypt | 1 | 1,099 | 1,099 | 1,099 |
| AES Encrypt | 1 | 1,063 | 1,063 | 1,063 |
| DVC | 3,537 | 9 | 2 | 66 |
| Fractals | 236 | 4,095 | 9 | 4,324 |
| Image Denoising | 1 | 44,925 | 44,925 | 44,925 |
| JPEG Decode | 1 | 431 | 431 | 431 |
| JPEG Encode | 2 | 582 | 421 | 744 |
| Radix Sort | 48 | 19 | 1 | 60 |
| Ray Tracing | 1,000 | 473 | 471 | 476 |
| RSA | 1 | >638,715 | >638,715 | >638,715 |
| SAD | 3 | 783 | 54 | 2,093 |
| SHA1 | 1 | 458 | 458 | 458 |

location for making spatial multitasking SM partitioning decisions. The GPU should still be responsible for scheduling individual threads to SMs within the partitioning constraints the OS or GPU runtime has set.

### 5.3.1 SM Reassignment

Repartitioning SMs among applications sharing the GPU requires the ability to take an SM from one application and assign the SM to another application. There are three approaches to handling the threads that are currently executing when an SM is reassigned:

- **Wait** for the threads to complete execution.
- **Terminate** the threads and restart them on another SM.
- **Migrate** the threads to another SM. I call this migration instead of preemption to differentiate it from preemptive multitasking. In GPU preemptive multitasking,

threads from *all* SMs are preempted on a context switch. In spatial multitasking with thread migration, threads from a *subset* of SMs are preempted when reassigning an SM to another application.

Each approach has advantages and disadvantages and the approaches can be combined into hybrid strategies where the best approach is chosen based on the situation. In all three approaches, actions (wait, terminate, or migrate) should minimally be applied to an entire warp because the 32 threads in a warp always issue instructions together. In practice it is probably better to apply actions to an entire thread block because threads within a block share state local to the SM, such as shared and local memory.

The simplest approach to implement is to wait for running threads to complete execution before assigning the SM to another application. This works well if the threads executing on the SM are short-running, because the SM can be reassigned relatively quickly. However with longer running threads, applications end up waiting for SMs to be assigned to them. In the worst case, where each application has a small number of long running threads, waiting for running threads to complete breaks down into cooperative multitasking. In the GT200 architecture, threads within a thread block can share state through a shared memory local to each SM. To avoid the need to migrate this state to another SM, one should allow all threads from any running thread block to complete execution. There is room for a small optimization here, if no running or complete threads from a block that is running access shared memory, the remaining threads in the block can be completed on another SM.

Terminating running threads allows SMs to be reassigned quickly. However, when terminating threads, the scheduler must be careful to ensure global-state has not been modified by any terminated threads. If a running thread has modified global state it should either be allowed to run to completion, migrated to another SM or the global-state changes

must be rolled-back and any threads that read these global changes must be rolled-back as well. Roll-back, or an atomic commit of global-state changes when a thread completes, can be implemented through the numerous mechanisms developed for transactional memory systems [70, 71]. However, a far simpler solution is to avoid terminating threads that have made global-state changes by migrating them or waiting for them to complete.

Migrating running threads to another SM allows work that has already been done to be saved while still allowing SMs to be reassigned relatively quickly. However, implementing thread migration is more complex than allowing threads to complete or terminating threads. When migrating a thread in the GT200 architecture the following state has to be saved:

- **Registers assigned to the thread.** The number of registers assigned to a thread is determined by the runtime environment when a kernel is launched. A thread can have a 4 to 128 registers assigned to it.

- **Shared memory.** Shared memory is allocated per thread block at compile time. A maximum of 16KB can be allocated by a thread block.

- **Program-counter of the thread.**

In the common case, when an SM is taken from an application, the remaining SMs assigned to the application will already be running the maximum number of thread blocks. Consequently, if the SM is to be reassigned immediately, then the thread state previously detailed must be saved rather than directly transferred to another SM. Alternatively, the migrate approach can be combined with the wait approach. Instead of immediately saving thread state, the scheduler can allow threads to continue to run until there are enough resources available on another SM to migrate the thread directly to the SM. As with using the wait approach alone, this combined approach breaks down into cooperative multitasking in the worst case.

Migrating threads introduces overhead that waiting for threads to complete does not, however migration can overcome this overhead when threads are long running and waiting for threads does not perform as well. A hybrid approach of wait, terminate, and migrate can take advantage of each approach's strength. When threads are near completion it should be best to wait for them to complete execution before reassigning an SM. If threads have just started and have not modified any global state they should be terminated. In all other cases, threads should be migrated to another SM. Exactly when to choose each approach is determined by the overhead associated with thread migration. If migration overhead is sufficiently small, a hybrid approach is unnecessary and migration alone is sufficient. If migration overhead is quite large, then a combination of wait and terminate is best.

In order to choose wait, terminate or migrate in a hybrid approach, two pieces of information are needed: how long a thread is expected to continue execution and if that thread has modified global state. The GPU can track global state modification by having a 1-bit flag per running thread that is initially cleared and set anytime a thread writes to global memory. This requires the addition of 1024-bits of state to each SM, one bit for each thread that can be in-flight on the SM. The GPU or runtime system can predict the length of a thread from other threads in the same kernel that have already completed execution. For the applications evaluated here there is very little variation in thread length within a kernel. The mean standard deviation of thread execution times is 10μs, across all simulated kernels. The largest observed difference in execution times of any two threads within a single kernel is 594μs. If a meaningful prediction cannot be made because no threads have completed execution then a default of either wait, terminate or migrate can be chosen.

### 5.3.2 Analytical Model

I have developed a simple mathematical model of two applications sharing the GPU using spatial multitasking. This model allows one to analyze when it is best to choose wait, migrate or terminate when migrating threads to another SM. For simplicity of analysis, in this model I assume both applications show an equal speedup of S over their performance in isolation when sharing the GPU. This assumption holds when using the Fair partitioning heuristic described later in Chapter 6, but is not a valid assumption for other heuristics. Due to the reduction in resources available to each application when sharing the GPU, S is likely to be less than 1.0, although due to the efficiencies of spatial multitasking it is likely to be greater than 0.5. I also assume the applications are uniform and this speedup is the same no matter when it occurs during the application's execution. Figures 5.6 and 5.7 show a generalized timeline of two applications sharing the GPU using either the wait, migrate, or terminate approach to thread migration. Figure 5.6 shows the case where application A finishes GPU execution before application B. Figure 5.7 shows the case where application A finishes GPU execution after application B. There are several other special cases that are not analyzed here because they are unlikely to occur commonly in practice or may lead to a situation where the choice to wait, migrate, or terminate has little impact on performance.

Figures 5.6 and 5.7 are annotated with several variables, in the form of $T_x$, that represent the amount of time spent in each phase of execution. Subscripts for T beginning with A represent times application A is executing in isolation, B represents times application B is running in isolation, and S represents times applications A and B are sharing the GPU via spatial multitasking. The labels are further explained here:

$T_{A1}$ — This is the time application A runs in isolation on the GPU before application B begins. During this time, I assume all of the GPU resources are available to application A and it is running at full speed.

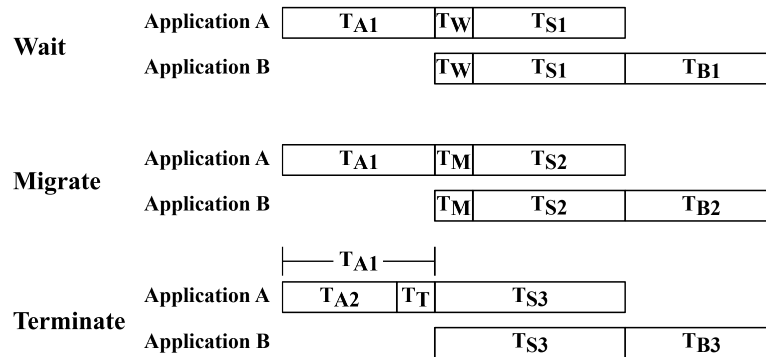



Figure 5.6: Timing of the wait, migrate, and terminate approaches to thread migration. Case 1: Application A finishes before application B.

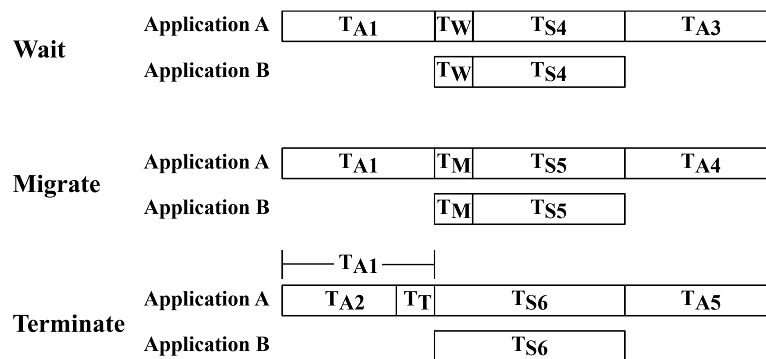



Figure 5.7: Timing of the wait, migrate, and terminate approaches to thread migration. Case 2: Application A finishes after application B.

$T_W$ — In the wait approach, the scheduler waits for all running threads on an SM to complete before assigning it to a new application. Time $T_W$ is the time after which application B starts that it must wait to be assigned SMs taken from application A. $T_W$ ends when the threads from application A that were being waited on complete execution. During $T_W$, I assume application A continues to run at full speed and application B does not run at all. Assuming application A runs with no slowdown during $T_W$ is idealized because as threads complete and new threads are not started application A would be using some of its SMs less efficiently. Therefore, this assumption favors wait over migrate.

$T_M$ — In the migrate approach $T_M$ represents the time application B is waiting for SMs to be assigned to it from application A. During this time, application A is not using a subset of its SMs because thread state is being migrated so some SMs can be assigned to application B. During this time, I assume application B does not run because it is waiting for SMs to be assigned to it. I assume application A shows a reduction in performance proportional to S because it is not using the SMs that are being preempted for thread migration, but still uses the remaining SMs assigned to it. As described earlier S is the speedup of the application when sharing the GPU using spatial multitasking.

$T_T$ — In the terminate approach $T_T$ represents the execution time that is lost from the threads that were terminated in application A to allow the scheduler to assign some SMs to application B. I assume application B begins execution immediately because the running threads from application A can be terminated immediately. I model the lost execution time of these threads by assuming application A has performance scaled by S during $T_T$. Note that application B does not start until after $T_T$ and I assume that application B still starts at the same time as in wait and migrate so $T_{A2} + T_T = T_{A1}$.

$T_{S1}$, $T_{S2}$, $T_{S3}$, $T_{S4}$, $T_{S5}$, $T_{S6}$ — These all represent the time that application A and application B are sharing the GPU normally using spatial multitasking. During this time I assume both applications have their performance scaled by S. Note that these times are not necessarily equal.

$T_{B1}$, $T_{B2}$, $T_{B3}$, $T_{A3}$, $T_{A4}$, $T_{A5}$ — These all represent the time after one of the applications has completed and the other is still running. During this time I assume the remaining application runs at full speed.

$T_{A2}$ — In the terminate approach, this is the time application A is running in isolation at full speed. During $T_T$ some of the execution is lost due to threads being terminated after $T_T$. Note that $T_{A2} + T_T = T_{A1}$.

Figure 5.8 shows the total execution time from the start of application A to the end of application B for wait, migrate, and terminate in case 1 (Figure 5.6). $T_W$ is the amount of execution time remaining in the threads running on the SMs that will be reassigned from application A to application B. For this example, I make the following assumptions:

- Application A and application B execute for 444K cycles in isolation. This is the average length of a CUDA kernel observed for the benchmarks presented in Section 3.1.
- Application A starts at 0 cycles and application B is arbitrarily chosen to start at 148K cycles.
- The time to preempt an SM and migrate threads ($T_M$) is 17.5K cycles. This length was chosen because the time for a context switch in NVIDIA's Fermi architecture is 25µs [4]. The NVIDIA GeForce GTX 480 has a shader clock of 700MHz, so a context switch should take 17.5K cycles. The Fermi context switch time is for an entire GPU context switch, so this may not be an accurate estimate of the time to perform a context switch on a subset of the GPU SMs as I use it here.

Figure 5.8 shows that when the amount of time remaining in the threads running on the SMs about to be reassigned is low ($T_W$ is near zero) it is best to wait for the threads to complete. When the threads have recently begun execution ($T_W$ is large) it is best to terminate them. Otherwise it is best to migrate threads.

To determine the best approach to take in the general case, I solve for the intersection of each of the approaches shown in Figure 5.8. I start by assuming the goal is to minimize the time from the first application starting to the last application completing. I define this

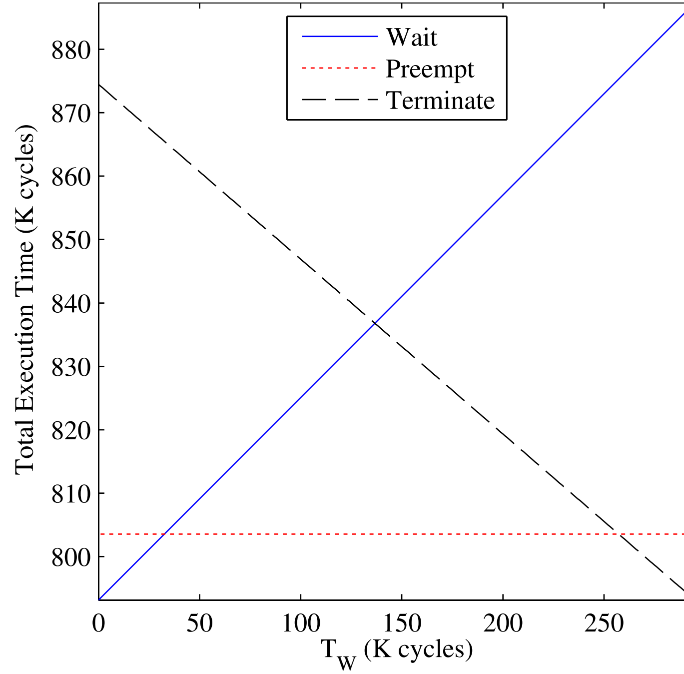



Figure 5.8: Example cost of the wait, migrate, and terminate approaches versus execution time remaining in application A's threads ($T_W$).

time as the cost and label it $C_x$ where $x$ is $W1$ for wait case 1, $M2$ for migrate case 2, etc. Figure 5.8 plots $C_{W1}$, $C_{M1}$, and $C_{T1}$. From Figures 5.6 and 5.7 the costs are:

$$\begin{aligned}
C_{W1} &= T_{A1} + T_W + T_{S1} + T_{B1} \\
C_{M1} &= T_{A1} + T_M + T_{S2} + T_{B2} \\
C_{T1} &= T_{A2} + T_T + T_{S3} + T_{B3} \\
C_{W2} &= T_{A1} + T_W + T_{S4} + T_{A3} \\
C_{M2} &= T_{A1} + T_M + T_{S5} + T_{A4} \\
C_{T2} &= T_{A2} + T_T + T_{S6} + T_{A5}
\end{aligned}$$

I denote the execution time of an application in isolation as $E_x(y)$, where $x$ is $A$ or B for application A or application B, respectively and $y$ is $W1$ for wait case 1, T2 for terminated

case 2, etc. Examining Figures 5.6 and 5.7 and using the assumption that an application's performance is scaled by S when using a subset of GPU resources, the execution times of application A in isolation are:

$$
\begin{aligned}
E_A(W1) &= T_{A1} + T_W + ST_{S1} \\
E_A(M1) &= T_{A1} + ST_{S2} \\
E_A(T1) &= T_{A2} + ST_T + ST_{S3} \\
E_A(W2) &= T_{A1} + T_W + ST_{S4} + T_{A3} \\
E_A(M2) &= T_{A1} + ST_{S5} + T_{A4} \\
E_A(T2) &= T_{A2} + ST_T + ST_{S6} + T_{A5}
\end{aligned}
$$

The execution times of application B in isolation are:

$$
\begin{aligned}
E_B(W1) &= ST_{S1} + T_{B1} \\
E_B(M1) &= ST_{S2} + T_{B2} \\
E_B(T1) &= ST_{S3} + T_{B3} \\
E_B(W2) &= ST_{S4} \\
E_B(M2) &= ST_{S5} \\
E_B(T2) &= ST_{S6}
\end{aligned}
$$

A full explanation of the derivation of $E_A$ and $E_B$ is presented in Appendix A. We are interested in finding the intersection of $C_x$ and $C_y$ for most of the $(x, y)$ pairs where $x$ and $y$ can be selected from $W1$, $M1$, $T1$, $W2$, $M2$, and $T2$. We are not interested in the intersection of the pairs $(W1, W2)$, $(M1, M2)$, and $(T1, T2)$, because we know the total execution time of

each application in isolation will remain unchanged. Combining this with the intersection of $C_x$ and $C_y$ we have three equations to work with:

$$
\begin{aligned}
C_x &= C_y \\
E_A(x) &= E_A(y) \\
E_B(x) &= E_B(y)
\end{aligned}
$$

Additionally, when solving for the intersection of wait or migrate with terminate we also know:

$$T_{A1} = T_{A2} + T_T$$

Let us assume $T_M$, the time it takes to migrate threads, is fixed and we are interested in solving for $T_W$ and $T_T$. $T_W$ corresponds to the amount of execution time remaining before completion of the threads on the SMs moving from application A to application B. $T_T$ corresponds to the execution time lost when threads are terminated in order to move SMs from application A to application B. By solving for $T_W$ and $T_T$ at the intersection of $C_x$ and $C_y$ we can determine, based on the predicted amount of execution time remaining in a thread when it is best to choose wait, migrate, or terminate.

Solving for $T_W$ at the intersection of $C_{W1}$ and $C_{M1}$ results in:

$$T_W = \frac{S^2}{2S-1} T_M$$

Solving for $T_T$ at the intersection of $C_{M1}$ and $C_{T1}$ results in:

$$T_T = \frac{S}{(S-1)^2} T_M$$

We see at the intersection of $C_{W1}$ and $C_{M1}$ that $T_W$ and $T_T$ are only dependent on the performance of the applications when executing on a subset of the GPU's SMs and the time it takes to migrate threads to another SM. This means if the length of migration is sufficiently small, then when the time remaining on the threads running on an SM to be reassigned is smaller than $\frac{S^2}{2S-1} T_M$ then wait should be chosen; if the time the threads have already been executing is less than $\frac{S}{(S-1)^2} T_M$ then terminate should be chosen; otherwise migrate should be chosen. If $T_M$ is sufficiently large then wait or terminate will always be a better choice than migrate. This occurs when either of the following are true:

$$\begin{aligned} T_M &> \frac{2S-1}{S} T_W \\ T_M &> \frac{(S-1)^2}{S} T_T \end{aligned}$$

When migration is sufficiently slow wait or terminate should always be selected. Solving for $T_W$ at the intersection of $C_{W1}$ and $C_{T1}$ results in:

$$T_W = \frac{(S-1)^2}{2S-1} T_T$$

This indicates for case 1 when choosing between wait or terminate, wait should be chosen when the amount of execution time remaining for the threads is less than $\frac{(S-1)^2}{2S-1} T_T$; otherwise, terminate should be chosen.

Assuming the goal is to minimize the time from the start of the first application to the

completion of the last application, then inspection of Figure 5.6 reveals that $C_{W2}$ is always less than $C_{M2}$. Assuming $0 < S < 1$ then $C_{W2}$ is always less than $C_{T2}$ as well. This also assumes that both $T_M$ and $T_T$ are greater than zero. The speedup must always be greater than zero ($S > 0$) and it is unlikely that applications would perform better when sharing the GPU rather than running in isolation, so S should be no more than one. Proofs of $C_{W2} < C_{M2}$ and $C_{W2} < C_{T2}$ are presented in Appendix B. In case 2, the scheduler should always choose wait over preempt or terminate.

When mixing case 1 and case 2, the choice between wait, preempt, and terminate is not as clear. The following are the solutions for these cases:

$$
\begin{aligned}
C_{W1} &= C_{M2} & T_W &= \frac{-S}{2S-1}T_M + T_{A4} \\
C_{M1} &= C_{W2} & T_W &= \frac{S}{2S-1}T_M - T_{A3} \\
C_{M1} &= C_{T2} & T_T &= \frac{S}{(S-1)^2}T_M - \frac{2S-1}{(S-1)^2}T_{A5} \\
C_{T1} &= C_{M2} & T_T &= \frac{S}{(S-1)^2}T_M + \frac{2S-1}{(S-1)^2}T_{A4} \\
C_{W1} &= C_{T2} & T_W &= \frac{(S-1)^2}{2S-1}T_T + T_{A5} \\
C_{T1} &= C_{W2} & T_W &= \frac{(S-1)^2}{2S-1}T_T - T_{A3}
\end{aligned}
$$

When comparing wait, preempt, and terminate with just case 1 or just case 2, the best choice could be made with the predicted length of the currently executing threads, the amount of time migration takes and the performance of the applications when sharing the GPU. When the difference in execution times between wait, migrate, and terminate require the comparison of case 1 and case 2, then the scheduler must also predict the length of the GPU kernels in order to choose the best approach.

Predicting the length of GPU kernels is likely to be more difficult than predicting the

length of individual threads. When predicting the length of a thread there are usually other threads from the same kernel whose execution time can be used as a guide. It is common for GPU kernels to be executed only once or with different inputs each time causing more variation in runtimes. For this reason, I recommend using only the information from case 1.

Assume there are several SMs to be taken from application A and assigned to application B. There are threads from application A currently executing on these SMs. The scheduler must decide whether to wait for these threads to complete before assigning the SMs to application B, migrate the threads to different SMs, or terminate the threads and restart them on other SMs. If the scheduler predicts the time to complete the execution of the threads is at most $\frac{S^2}{2S-1}T_M$ then it should wait for the threads to complete. If the amount of time the threads have already executed for is at most $\frac{S}{(S-1)^2}T_M$ then the scheduler should terminate the threads; otherwise it should migrate the threads. There should also be a threshold on the amount of time the scheduler waits for threads to complete. If this threshold is crossed and the scheduler is still waiting for the threads to complete then it should migrate the threads to another SM. This ensures a faulty or malicious application cannot monopolize the GPU. Finally, if the scheduler is unable to predict the length of the threads it should migrate the threads.

The length of time from the start of the first application to the completion of all applications is chosen as the performance metric for this analysis because it results in a system of linear equations that are easy to analyze and understand. The sum of execution times of the applications also shares these properties. However, these performance metrics may not be the best choice for optimization. Another goal could be to maximize the sum of the speedup of each application to the $N^{th}$ power. With high values of N this ensures a relatively even performance boost for each application. Unfortunately, even if $N = 1$ is chosen speedup adds non-linearity to the analytical model making it more difficult to solve the system of equations and analyze the results.

## 5.4 Conclusion

I compared spatial multitasking to cooperative multitasking through simulation. Spatial multitasking allows multiple applications to simultaneously share the GPU by partitioning GPU resources among applications, rather than or in addition to, time multiplexing applications, as is done by cooperative and preemptive multitasking. Simulation results indicate that spatial multitasking has the potential to offer significant performance benefits compared to cooperative multitasking by executing applications in parallel, showing average speedups of 1.14, 1.22, and 1.30 for two, three, and four applications sharing the GPU, respectively. This implementation used a simple even distribution of SMs among applications. Chapter 6 explores alternative partitioning heuristics. By allowing applications to share the GPU, spatial multitasking out-performs cooperative multitasking by using available GPU resources more efficiently.

# 6 SM PARTITIONING

## 6.1 Methodology

In the previous section I analyzed the performance advantages of spatial multitasking over cooperative multitasking using a simple even-split of GPU SMs among applications. In this section I compare the following alternative SM partitioning heuristics:

**Oracle Best** performs an exhaustive search of SM partitionings and chooses the one that maximizes speedup over cooperative multitasking. This heuristic is impractical to implement in real systems, but provides an upper bound on performance. Due to constraints on real world simulation time I was unable to simulate Oracle scheduling for more than two applications sharing the GPU. I calculated, with a 120 node cluster, it would take over 400 days to complete the simulations necessary for oracle scheduling of 3 applications.

**Oracle Worst** performs an exhaustive search of SM partitionings and chooses the one that minimizes speedup over cooperative multitasking. This heuristic is impractical to implement outside of simulation but provides a lower bound on performance.

**Even** distributes SMs as evenly as possible among applications. When the number of SMs is not divisible by the number of applications, the assignment is made arbitrarily with a near-even split. For example, in the case of four applications sharing a 30 SM GPU two applications receive seven SMs and two receive eight.

**Smart Even** is the same as Even except it only gives an application the maximum number of SMs it can use based on the number of thread blocks in the program. SMs that go unused by an application are divided evenly among the other applications sharing the GPU. RSA is the only application evaluated where partitioning differs between

Even and Smart Even. In Smart Even, RSA is assigned four SMs because it has four thread-blocks for the input data used here.

**Packed** is similar to Smart Even except that it gives an application the minimum number of SMs required to have all thread-blocks in-flight simultaneously. In Packed, RSA is assigned one SM because all four RSA thread-blocks can be simultaneously in-flight on one SM.

**Rounds** attempts to assign SMs such that there are fewer idle SMs near the completion of a kernel. In this simple implementation of spatial multitasking in simulation, SMs cannot be reassigned to another application after the initial assignment. Thus, when a GPGPU kernel has nearly completed execution, SMs can be idle because there are no new thread blocks to assign to them. I have observed that most thread blocks from a single kernel take the same amount of time to execute, since thread blocks tend to execute in rounds where groups of thread blocks start and complete execution at nearly the same time. By knowing the number of thread blocks the GPU would assign to each SM (it can be different for each kernel) Rounds can predict how many rounds it would take to execute a kernel. For example assume an application can run eight thread blocks on an SM simultaneously. If the application is assigned 15 SMs then it can execute a total of 120 thread blocks simultaneously. If the application has 256 thread blocks then it would take ceiling$(256 \div 120) = 3$ rounds to complete all execution. If the application is assigned 14 SMs then 112 thread blocks can execute simultaneously and it still takes three rounds to complete. The Rounds heuristic starts with an even-split of SMs among applications and calculates the number of rounds each application would take to execute. After this, Rounds finds the minimum number of SMs an application can be assigned without increasing the number of rounds the application would take to execute over an even-split of SMs. Rounds then

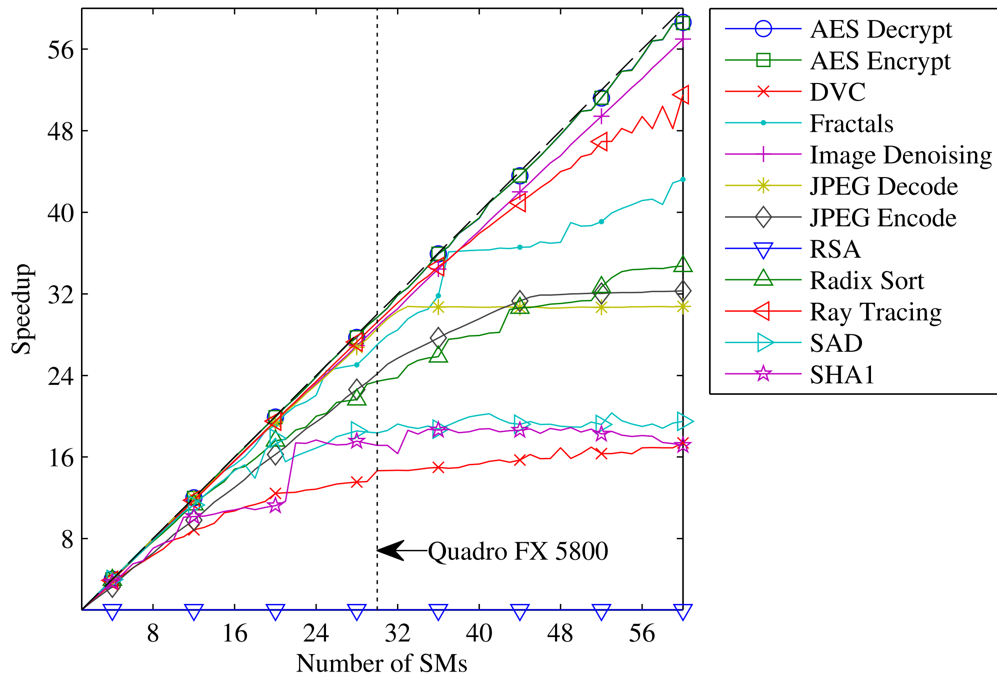



Figure 6.1: Speedup of GPU computation versus number of SMs. Results are normalized to one SM. The dashed line represents linear speedup.

finds which applications would benefit from adding SMs assuming other applications receive the minimum number of SMs just calculated. Finally, Rounds chooses the partitioning that results in the minimum total number of rounds summed over all of the applications. In the case of ties, Rounds chooses the partitioning that is closest to an even-split. To summarize, this heuristic attempts to minimize the total number of rounds executed among all the applications while ensuring that no application takes more rounds to execute than it would require with a simple even-split.

**Profile** uses the information from Figure 6.1 (reproduced from Chapter 4) to choose the best SM partitioning. An implementation of this heuristic requires that applications are profiled in isolation before the heuristic can partition SMs. This would typically be done at install or compile time. For some applications the performance versus the number of SMs assigned is dependent on the input data. For these cases, a more

detailed profiling heuristic may be necessary. For this research, I used the same input for profiling the applications and evaluating the partitioning heuristics. This decision is likely to favor Profile over the other heuristics. The partitioning that is chosen by this heuristic maximizes the following function:

$$\sum_{i=1}^{N} S(i,j)^{\frac{1}{N}} \tag{6.1}$$

where $S(i,j)$ is the speedup of application $i$ depicted in Figure 6.1 when $j$ SMs are assigned to the application and $N$ is the number of applications sharing the GPU. Other cost functions were evaluated for weighting profiling results, particularly many alternatives to $\frac{1}{N}$ were evaluated, but the function presented was the best that was evaluated.

**Fair** attempts to equalize the performance loss applications experience sharing the GPU via spatial multitasking compared to running in isolation. I evaluate performance loss using two methods. In the case where two applications share the GPU I use an exhaustive search of all SM partitionings to determine actual performance loss versus isolation and select the partitioning that minimizes the difference in performance loss between the two applications. An exhaustive search is not practical in a real system so the Fair heuristic attempts to predict application performance loss using the profiling information presented in Figure 6.1. Using this data I evaluate the Fair heuristic for two, three, and four applications sharing the GPU.

**Blocks** assigns SMs to applications proportional to the number of thread-blocks the applications have in a kernel. In the cases where an application had several kernels with different thread-block counts, I used the weighted average of thread-blocks across kernels as detailed in Table 6.1 reproduced from Section 3.1. The weighting

Table 6.1: Weighted average application thread configuration and blocks assigned per SM as used by the partitioning heuristics.

| **Application** | **Blocks** | **Threads per Block** | **Blocks per SM** |
|---|---|---|---|
| AES Decrypt | 4,097 | 256 | 5 |
| AES Encrypt | 4,097 | 256 | 3 |
| DVC | 112 | 216 | 2 |
| Fractal Generation | 512 | 64 | 8 |
| Image Denoising | 110,592 | 64 | 8 |
| JPEG Decode | 13,824 | 64 | 8 |
| JPEG Encode | 73,694 | 64 | 8 |
| RSA | 4 | 32 | 3 |
| Radix Sort | 357 | 250 | 4 |
| Ray Tracing | 2,048 | 128 | 8 |
| SAD | 4,354 | 66 | 8 |
| SHA1 | 65 | 64 | 3 |

is percentage of execution time spent in each kernel. If dynamic SM assignment were available in my implementation of spatial multitasking on GPGPU-Sim Blocks could re-evaluate SM partitioning at the start of each kernel and likely show better performance. The formula Blocks uses for calculating the number of SMs application $i$ receives is presented below. Blocks rounds $SMs_i$ towards zero and always gives an application at least one SM. The last application to be assigned SMs is simply given the remaining unassigned SMs.

$$SMs_i = TotalSMs \frac{Blocks_i}{\sum_{i=1}^{N} Blocks_i} \tag{6.2}$$

**Threads-per-block** is the same as Blocks except it uses the number of threads-per-block in an application instead of the number of thread-blocks.

**Threads** is the same as Blocks except it uses total number of threads in an application (thread-blocks × threads-per-block) instead of the number of thread-blocks.

Table 6.1 details, for each application, the weighted average thread configuration and most likely (by percentage of execution time in isolation) number of thread blocks executing simultaneously per SM. Average blocks and threads per block are calculated as the weighted arithmetic mean (Equation 6.3) of blocks and threads per block, respectively, where $i$ is the current kernel, $n$ is the total number of kernels, $w_i$ is the percent of execution time spent in kernel $i$, and $x_i$ is the number of blocks or threads per block in kernel $i$. The Smart Even, Packed, Rounds, Blocks, Thread-per-block, and Threads partitioning heuristics use the information in Table 6.1.

$$\bar{x} = \sum_{i=1}^{n} w_i x_i \tag{6.3}$$

All of these experiments use a static partitioning of GPU SMs among applications. The SM partitioning is decided at the beginning of each simulation and never changed. In a real implementation, SMs should be reassigned among applications dynamically to allow SMs to be assigned to new GPGPU kernels and reclaimed from completed kernels. Dynamic partitioning is likely to result in even greater performance improvements for spatial multitasking due to more efficient use of GPU SMs. Dynamic partitioning may also require modifications to the partitioning heuristics presented here.

I obtained results for two applications sharing the GPU by simulating all possible partitionings of all combinations of applications presented in Section 3.1. I used the baseline NVIDIA Quadro FX 5800 GPU described in Section 2.2 and assumed no SMs could be left idle, i.e. if application A is assigned 25 of the 30 SMs in the GPU then application B is assigned the remaining 5 SMs. This enabled me to quickly explore partitioning heuristics for the two application space. Once the simulations completed, evaluating a partitioning heuristic was simply a matter of writing a python script to choose the SM partitioning for a given pair of applications and then extract the pre-existing simulation

results. Unfortunately the number of simulations required to exhaustively explore three or four applications sharing the GPU is prohibitively high for the compute resources available to us. Therefore, after exploring heuristics for partitioning SMs among two applications sharing the GPU, I selected the most promising heuristics and selectively simulated the necessary SM partitionings for all combinations of three and four applications sharing the GPU.

In this evaluation I have used the same methodology described previously in Section 5.1. I simulate applications sharing the GPU via spatial multitasking for 5M GPU cycles. I then measure the number of instructions each application executes in spatial multitasking and simulate the same number of instructions in cooperative multitasking. While this ensures a fixed amount of work is compared between spatial and cooperative multitasking simulations it has a potentially significant drawback when evaluating spatial multitasking partitioning heuristics. Partitioning heuristics can significantly favor one application over another, starving out execution of applications poorly suited for spatial multitasking in favor of applications well suited for spatial multitasking. This means heuristics that select extreme distributions of SMs, i.e. 29 SMs for one application and 1 SM for another, may have their performance overstated. Analysis of the results for two applications sharing the GPU indicate this happens twice for Rounds, once for Profile, once for Fair, and once for Oracle out of 78 combinations each. So for these heuristics favoring applications well suited for spatial multitasking is unlikely to have a significant impact on the results.

## 6.2 Evaluation

Table 6.2 shows the performance of each partitioning heuristic for all combinations of two applications sharing the GPU. Based on these results I chose to further evaluate Even, Smart Even, Rounds, Profile, and Fair (Fair Profile) when three or four applications share

Table 6.2: Speedup of several SM partitioning heuristics when all combinations of two applications share the GPU.

| **Heuristic** | **Geo Mean** | **Min** | **Max** |
|---|---|---|---|
| Oracle Best | 1.19 | 1.00 | 2.00 |
| Oracle Worst | 1.00 | 0.89 | 1.97 |
| Even | 1.14 | 0.99 | 2.00 |
| Smart Even | 1.16 | 0.99 | 2.00 |
| Packed | 1.16 | 0.99 | 2.00 |
| Rounds | 1.16 | 0.99 | 2.00 |
| Profile | 1.17 | 0.99 | 2.00 |
| Fair Exhaustive | 1.16 | 0.99 | 2.00 |
| Fair Profile | 1.17 | 0.97 | 2.00 |
| Blocks | 1.13 | 0.99 | 2.00 |
| Threads-per-block | 1.13 | 0.99 | 2.00 |
| Threads | 1.12 | 0.99 | 2.00 |

the GPU. Although Even performs poorly relative to the other heuristics, I chose to further evaluate it due to its extreme simplicity. It also served as the logical starting partitioning heuristic for the initial evaluation of spatial multitasking presented in Chapter 5. Smart Even and Packed both provide obvious improvements to Even. I chose not to evaluate Packed further because it is nearly the same as Smart Even and shows little performance difference. Rounds, Profile, and Fair are all unique and provided compelling performance so I evaluated them further. Block, Threads-per-block, and Threads all performed worse than Even, the simplest heuristic, so I chose not to evaluate them any further.

Figure 6.2 compares the performance of the SM partitioning heuristics using the baseline NVIDIA Quadro FX 5800 configuration. Geometric mean speedup of spatial multitasking across all combinations of applications is shown with the results normalized to cooperative multitasking. Smart Even improves performance over Even because SMs are not wasted on applications that will not use them. However, the difference in performance between Smart Even, Fair, Rounds, and Profile is insignificant for this GPU configuration. When two applications share the GPU, these simple partitioning heuristics approach the opti-

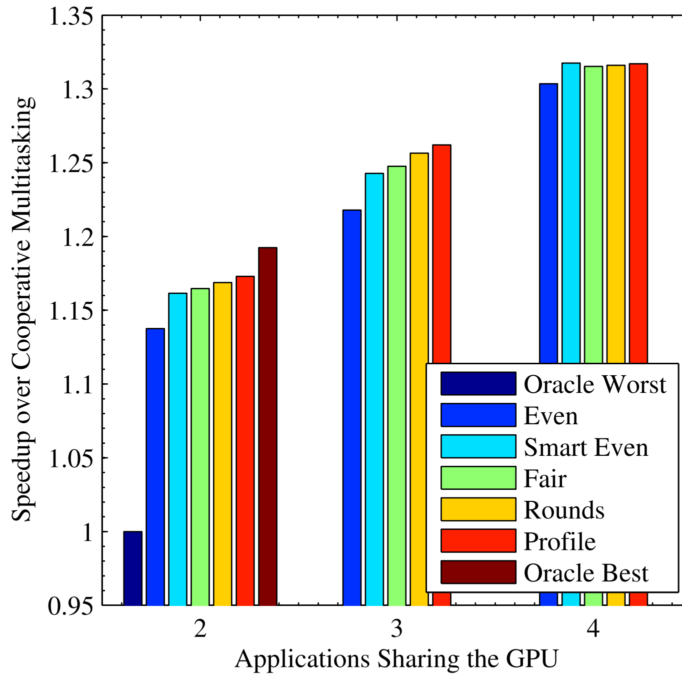



Figure 6.2: Average speedup of spatial multitasking over cooperative multitasking for several SM partitioning heuristics.

mal Oracle Best performance. The average performance of Oracle Worst is close to the performance of cooperative multitasking, which suggests that even with poor partitioning heuristics spatial multitasking can outperform cooperative multitasking.

Figures 6.3 and 6.4 plot the performance of several SM partitioning heuristics as the total number of SMs in the GPU is varied. These results indicate that as fewer SMs are available per application the best scheduler to choose changes. When there are a large number of SMs available for each application, Smart Even performs nearly as good as Fair, Rounds, and Profile. Profile and Fair require an extra profiling step to be run for each application at install time and Rounds is complex to implement. Smart Even is an attractive solution because it is very simple to implement and performs nearly as well as Rounds and Profile when there are many SMs available per application. However as resources become more constrained Smart Even does not perform as well as Fair, Rounds, and Profile. In this case

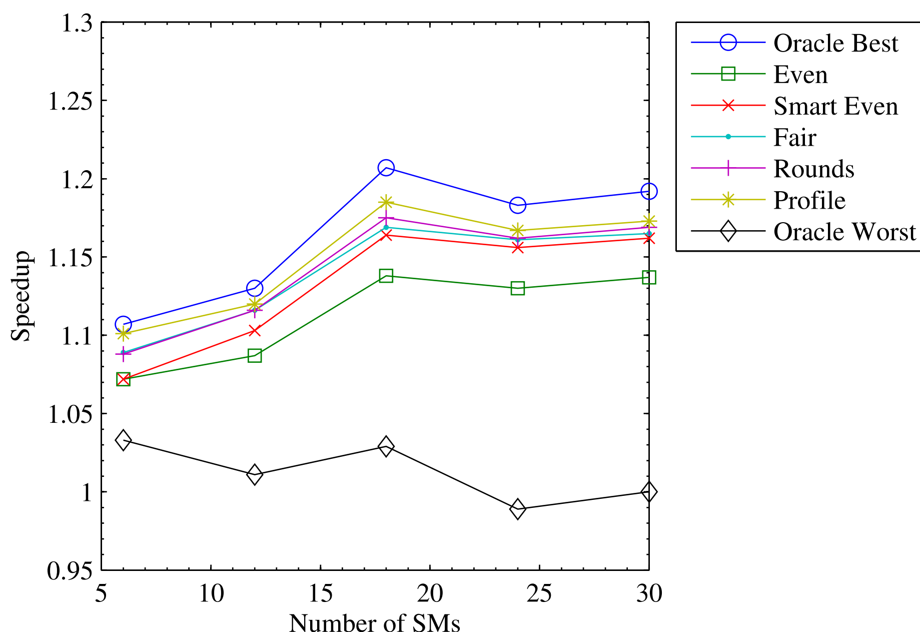



Figure 6.3: Average speedup of spatial multitasking compared to cooperative multitasking for several SM partitioning heuristics as the total number of SMs in the GPU is varied. Two applications share the GPU.

Rounds would be the best heuristic to choose because it performs just as well as Fair and Profile, but does not require an extra profiling step at install time. If profiling information is available then Fair is a good choice for partitioning heuristic. Fair performs roughly equal to Rounds and Profile, but results in an even performance degradation across all applications when sharing the GPU. Profile favors applications that are more likely to improve total system performance in spatial multitasking at the cost of other applications.

Looking beyond average performance, I have observed that in individual cases the best performing heuristics varies. This indicates a hybrid of Smart Even, Fair, Rounds, and Profiling could outperform any of the heuristics on their own.

Chapter 5 presented the speedup of spatial multitasking over cooperative multitasking for several specific combinations of applications. These combinations are examined further

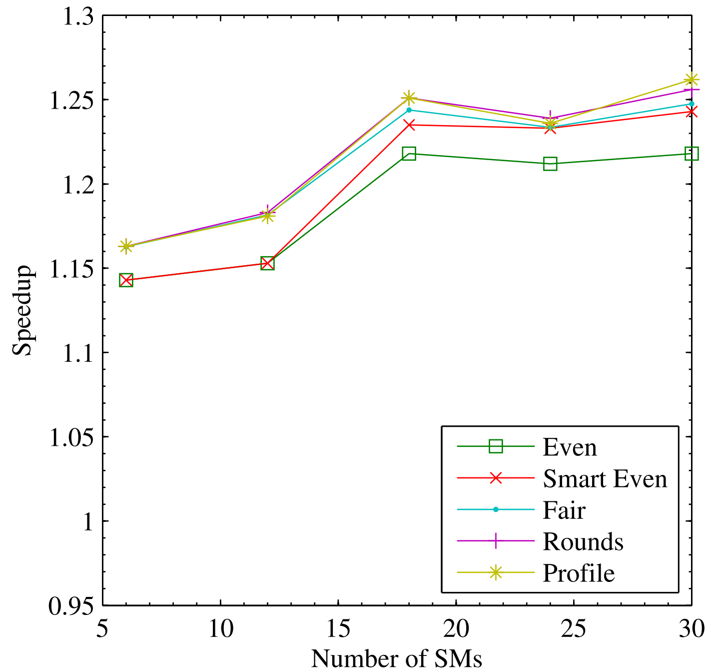



Figure 6.4: Average speedup of spatial multitasking compared to cooperative multitasking for several SM partitioning heuristics as the total number of SMs in the GPU is varied. Three applications share the GPU.

Table 6.3: Distribution of SMs for several specific combinations of applications.

| **Application** | **Oracle Worst** | **Smart Even** | **Fair** | **Rounds** | **Profile** | **Oracle Best** |
|---|---|---|---|---|---|---|
| DVC | 29 | 15 | 14 | 14 | 8 | 8 |
| SHA1 | 1 | 15 | 16 | 16 | 22 | 22 |
| AES Encrypt | 1 | 15 | 15 | 15 | 16 | 25 |
| Image Denoising | 29 | 15 | 15 | 15 | 14 | 5 |
| JPEG Decode | 2 | 15 | 16 | 15 | 17 | 20 |
| Radix Sort | 28 | 15 | 14 | 15 | 13 | 10 |
| JPEG Encode | 1 | 15 | 18 | 15 | 14 | 25 |
| SAD | 29 | 15 | 12 | 15 | 16 | 5 |

in Tables 6.3 and 6.4. Table 6.3 lists the distribution of SMs selected by each heuristic and Table 6.4 presents the speedup of spatial multitasking measured over cooperative multitasking. For the applications listed in Tables 6.3 and 6.4 Smart Even always selects

Table 6.4: Speedup of spatial multitasking over cooperative multitasking for several specific combinations of applications.

| **Application** | **Oracle Worst** | **Smart Even** | **Fair** | **Rounds** | **Profile** | **Oracle Best** |
|---|---|---|---|---|---|---|
| DVC<br>SHA1 | 0.99 | 1.16 | 1.09 | 1.09 | 1.23 | 1.23 |
| AES Encrypt<br>Image Denoising | 1.00 | 1.01 | 1.01 | 1.01 | 1.01 | 1.01 |
| JPEG Decode<br>Radix Sort | 0.99 | 1.06 | 1.07 | 1.06 | 1.07 | 1.08 |
| JPEG Encode<br>SAD | 1.00 | 1.05 | 1.05 | 1.05 | 1.04 | 1.08 |

and even split of SMs among the applications. Rounds and Fair generally also select and even or near even split. Profile varies further from and even split and Oracle Best strays even more. Oracle Worst selects very extreme distributions of SMs.

In three of the four cases presented in Table 6.4 there is little performance difference between all the heuristics except Oracle Worst. In the case of DVC and SHA1, Profile and Oracle Best greatly outperform the other heuristics. As discussed in Chapter 5, I have classified both DVC and SHA1 as problem size bound applications. Two problem size bound applications sharing the GPU is most likely to show the greatest speedup in spatial multitasking and as shown in Table 6.4 this is when the partitioning heuristic has the greatest impact on performance.

## 6.3 Conclusion

In Chapter 5 I showed spatial multitasking outperforms cooperative multitasking with a simple even split of SMs among applications, due to more efficient use of GPU resources. Here I have shown alternative partitioning heuristics which are able to further improve performance of spatial multitasking. The Rounds heuristic uses GPGPU kernel thread

configuration to optimize the number of SMs each application receives. The Profile heuristic uses information on performance scaling in isolation obtained at install or compile time to predict application performance when sharing the GPU via spatial multitasking. The Fair heuristic attempts to balance the loss in performance applications experience when sharing the GPU via spatial multitasking. All three of these heuristics outperform a simple even split of SMs among applications sharing the GPU via spatial multitasking.

# 7 QUALITY-OF-SERVICE

In previous chapters I explored the performance of spatial multitasking versus cooperative multitasking under the assumption that all applications benefit from more compute time and resources. However, some applications instead have a fixed level of required performance, and additional compute time and/or resources are not useful. Video playback, for example, generally proceeds at a fixed frame rate. If performance is sufficient to decode video at the required frame rate, then further performance improvements do not benefit the video playback application. I will refer to this class of applications as *QoS applications* and other applications as *best-effort*. When QoS and best-effort applications share the GPU, total system performance is maximized by providing QoS applications with the minimum compute time and resources they need to meet their requirements, and giving the remaining time and resources to the best-effort applications. In this chapter, I explore the performance benefits of spatial multitasking compared to cooperative multitasking when QoS and best-effort applications share the GPU. I also examine the cost of statically reserving enough SMs for QoS applications to meet performance requirements for the worst case versus adaptively changing the SM reservation based on what applications are co-scheduled with the QoS application.

## 7.1 Evaluation

Cooperative multitasking assigns all resources to a QoS application for a sufficient *time* that the application can meet its QoS goals. On the other hand, spatial multitasking assigns a sufficient number of *resources* to meet the application's goals. Future work could combine temporal and spatial multitasking to free those resources when not needed.

In the following experiments, all QoS applications meet their goals. The difference

between the "performance" of cooperative vs. spatial multitasking is based on the performance achieved by the best-effort applications that use the remaining time or resources. Thus, all reported speedups are the geometric mean of the best-effort applications.

The methodology I use in this chapter to compare spatial and cooperative multitasking is similar to previous chapters. I have selected SAD as the QoS application. SAD is commonly used in video encoding which, when done on-line for encoding video capture, typically has a fixed frame-rate making it a natural fit for QoS. SAD does not scale as well as other applications as GPU resources are increased, so selecting SAD as the focus of this evaluation may favor spatial multitasking over cooperative multitasking. In spatial multitasking SAD is assigned 5, 10, 15, 20, and 25 SMs to represent varying performance requirements SAD may have to meet. The remainder of the SMs are assigned to two best-effort applications sharing the GPU with SAD. I evaluate all combinations of the applications presented in Section 3.1 as best-effort applications, including more instances of SAD. I simulate this spatial multitasking configuration for 5M GPU cycles and measure the number of instructions each application executes. I then simulate the same combination of applications using cooperative multitasking for the same number of instructions. Speedup compares the number of GPU cycles it takes to execute the applications in cooperative multitasking versus the 5M GPU cycles simulated for spatial multitasking.

Reserving a fixed number of SMs for an application to meet its QoS targets under spatial multitasking is likely to slightly overshoot the desired performance target of the QoS application. In this methodology cooperative multitasking simulates the same number of instructions as spatial multitasking. This means cooperative multitasking is unnecessarily overshooting the QoS performance target as well. Consequently the performance of cooperative multitasking is slightly understated.

Figure 7.1 compares the performance of several partitioning heuristics when combined with QoS for SAD. The number of SMs assigned to SAD is shown on the x-axis, while

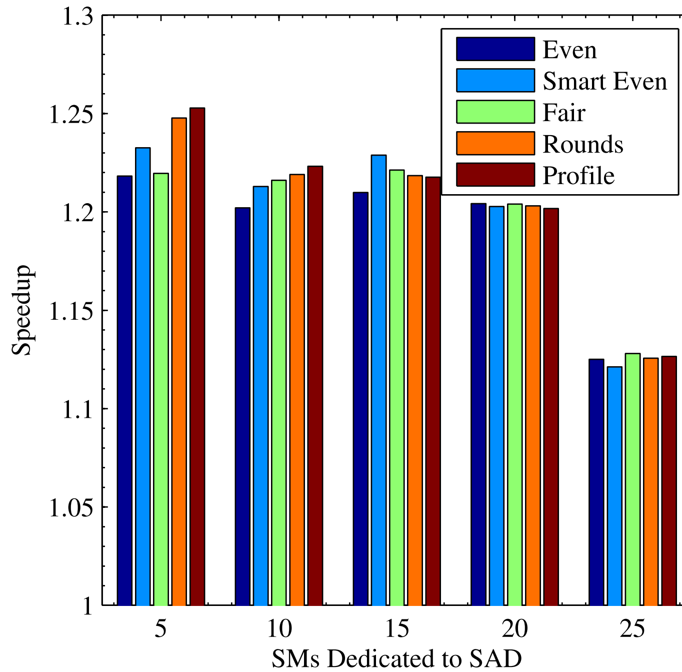



Figure 7.1: Speedup of spatial multitasking compared to cooperative multitasking, broken down by partitioning heuristic, when SAD is provisioned a fixed number of SMs to provide QoS.

the speedup of spatial multitasking compared to cooperative multitasking is shown on the y-axis. The same number of instructions are simulated for each application in both cooperative and spatial multitasking. This methodology compares the same level of QoS for SAD in both spatial and cooperative multitasking, with the difference being cooperative multitasking is time multiplexing applications to provide QoS while spatial multitasking is provisioning a fixed number of SMs to an application to provide QoS. Table 7.1 summarizes the results plotted in Figure 7.1.

Spatial multitasking performance relative to cooperative multitasking declines as the level of QoS required by SAD rises (more SMs provisioned to it), however spatial multitasking consistently outperforms cooperative multitasking. Rounds and Profile perform best overall for partitioning SMs among the best-effort applications. However, as the number of

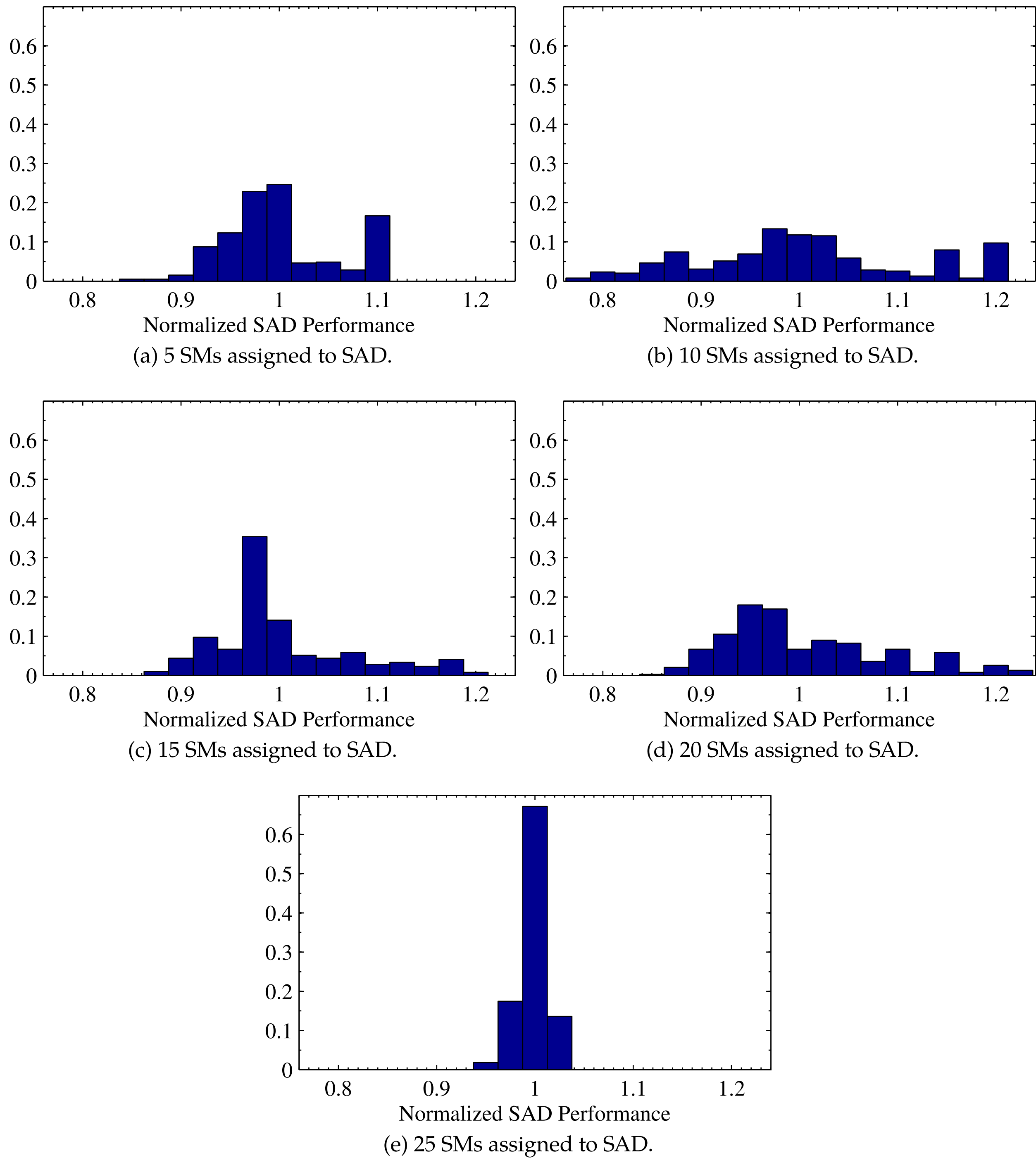
Table 7.1: Total system speedup using spatial multitasking compared to cooperative multitasking, broken down by partitioning heuristic, when QoS SAD shares the GPU with two best-effort applications.

| **SMs Reserved for SAD** | **Even** | **Smart Even** | **Fair** | **Rounds** | **Profile** |
|---|---|---|---|---|---|
| **5** | 1.22 | 1.23 | 1.22 | 1.25 | 1.25 |
| **10** | 1.20 | 1.21 | 1.22 | 1.22 | 1.22 |
| **15** | 1.21 | 1.23 | 1.22 | 1.22 | 1.22 |
| **20** | 1.20 | 1.20 | 1.20 | 1.20 | 1.20 |
| **25** | 1.12 | 1.12 | 1.13 | 1.13 | 1.13 |

SMS provisioned for SAD rises there is less variation among the partitioning heuristics.

The performance of SAD varies depending on what applications it is co-scheduled with. If the number of SMs reserved for SAD is fixed without consideration for the applications it is co-scheduled with, then the reservation must be over-conservative to account for the worst-case. Figure 7.2 plots the distribution of normalized performance for QoS SAD when co-scheduled with different combinations of two best-effort applications. Performance (total instructions simulated) is normalized to geometric mean performance. These plots provided insight into the level of performance variation expected when SAD is co-scheduled with other applications. This performance variation is caused by interference in shared hardware, such as the memory system and interconnect, when sharing the GPU via spatial multitasking. SAD performance can vary significantly depending on the applications it shares the GPU with. Unfortunately, this means SAD's SM reservation would have to be significantly over-provisioned to guarantee a target performance if we fail to consider which applications are co-scheduled with it. Over-provisioning SMs for QoS applications to guarantee performance targets results in fewer SMs assigned to best-effort applications, resulting in reduced performance of best-effort applications.

Table 7.2 summarizes the observed worst case variation in performance for each application when two applications, one QoS and one best-effort, are sharing the GPU via spatial




(a) 5 SMs assigned to SAD.

(b) 10 SMs assigned to SAD.

(c) 15 SMs assigned to SAD.

(d) 20 SMs assigned to SAD.

(e) 25 SMs assigned to SAD.

Figure 7.2: Distribution of normalized performance of QoS SAD when sharing the GPU with two best-effort applications via spatial multitasking. Performance is normalized to the geometric mean performance for the specified number of SMs assigned to SAD.

Table 7.2: Maximum observed performance degradation when two applications share the GPU via spatial multitasking.

| Application | Maximum Perf. Loss |
|---|---|
| AES Decrypt | 6.8% |
| AES Encrypt | 8.2% |
| DVC | 29.7% |
| Fractal Generation | 4.9% |
| Image Denoising | 3.7% |
| JPEG Decode | 42.8% |
| JPEG Encode | 43.9% |
| RSA | 75.9% |
| Radix Sort | 21.0% |
| Ray Tracing | 11.9% |
| SAD | 34.2% |
| SHA1 | 43.1% |

multitasking. Worst case performance variation is defined as the maximum percentage loss in performance for an application in spatial multitasking compared to running in isolation with the same number of SMs. Several applications, such as Image Denoising, show very little performance variation under spatial multitasking, indicating they are insensitive to applications co-scheduled on the GPU via spatial multitasking. These applications are well suited to a static SM provisioning method. Other applications, such as RSA, show a very large potential performance loss in spatial multitasking depending on the application with which they are co-scheduled. To improve system performance, when reserving SMs for these applications co-scheduled applications should be considered.

Next, I evaluate the benefits of choosing the number of SMs to reserve for a QoS application based on what applications it is co-scheduled with. To evaluate this I look at every combination of two applications sharing the GPU where one is a QoS application and the other best-effort. I perform an exhaustive search of each possible SM partitioning to determine the minimum number of SMs required for the QoS application to meet its target performance. The remaining SMs are given to the best-effort application. I

Table 7.3: Geometric mean speedup of dynamic QoS compared to static when two applications share the GPU broken down by application. Speedup is listed for the applications sharing the GPU with the specified application. i.e. when QoS is provided for the listed application how much faster is dynamic QoS for the best-effort applications. The QoS target is a minimum of 75% of the performance in isolation.

| **Application** | **Speedup** | **Min SMs** | **Max SMs** |
|---|---|---|---|
| AES Decrypt | 1.00 | 23 | 23 |
| AES Encrypt | 1.00 | 23 | 23 |
| DVC | 1.23 | 18 | 22 |
| Fractal Generation | 1.11 | 22 | 23 |
| Image Denoising | 1.00 | 23 | 23 |
| JPEG Decode | 1.07 | 22 | 23 |
| JPEG Encode | 1.02 | 22 | 23 |
| RSA | 2.81 | 1 | 23 |
| Radix Sort | 1.06 | 20 | 22 |
| Ray Tracing | 1.00 | 23 | 23 |
| SAD | 1.35 | 16 | 22 |
| SHA1 | 1.00 | 22 | 22 |
| Geometric Mean | 1.16 | | |

arbitrarily set the target performance for the QoS applications to 75% of their performance in isolation. I simulate this dynamic QoS partitioning for 5M GPU cycles and measure the number of instructions executed by the best-effort application. Instructions simulated by the QoS applications are measured only to ensure they meet or exceed the target level of performance. I compare this dynamic approach to a static approach that does not consider which applications are co-scheduled and conservatively reserves enough SMs for the QoS application to meet performance targets regardless of co-scheduling.

Table 7.3 shows the potential speedup of dynamic QoS over static QoS using spatial multitasking. The table breaks down the speedup by application. The application is guaranteed a performance level of 75% of uncontested performance when running in isolation. Table 7.3 also lists the minimum and maximum number of SMs each application requires to reach the 75% target performance level. The required number of SMs varies based on what application is co-scheduled with the listed application.

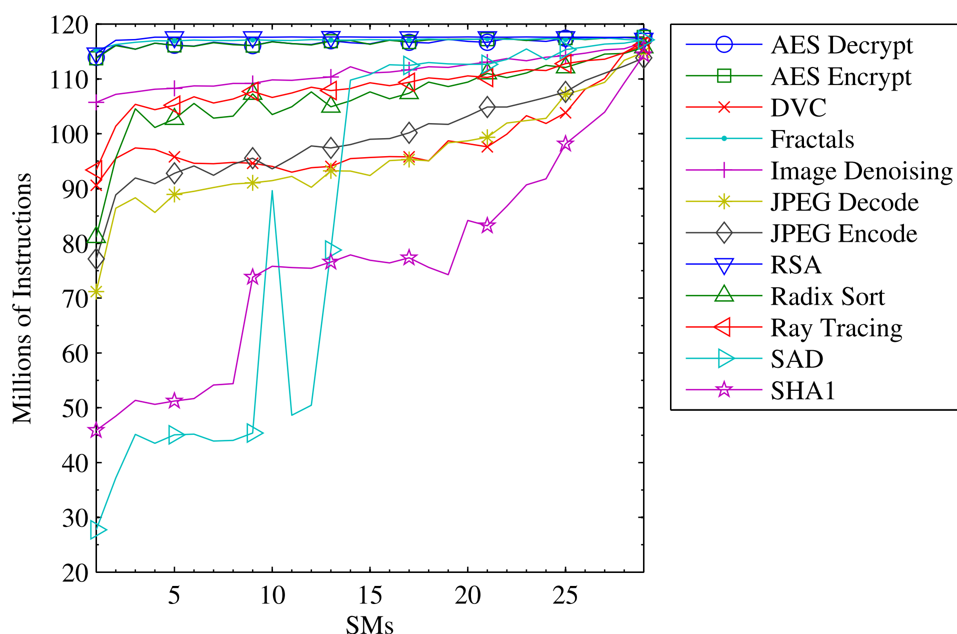



Figure 7.3: Performance of RSA when sharing the GPU with one application via spatial multitasking. The x-axis is the number of SMs RSA receives and the y-axis is millions of instructions executed by RSA. Each application RSA shares the GPU with is plotted separately.

The performance of some applications, such as AES, is not affected by the application co-scheduled with it. Consequently, with these applications there is no benefit to a dynamic QoS scheme over a static scheme. Other applications, such as RSA, show great variation in performance depending on what application is co-scheduled with them. For these applications, system performance is greatly improved with a dynamic QoS scheme.

Figure 7.3 plots the performance of RSA based on what application is co-scheduled with it and how many SMs are assigned to RSA. Recall that RSA is only able to make use of a maximum of four SMs when running in isolation. Despite this limitation, when co-scheduled with certain applications the performance of RSA continues to rise as the number of SMs it receives grows well beyond four. This is because RSA is very sensitive to interference in the interconnect and memory system. When RSA is given more SMs

the application it is co-scheduled with receives less SMs. This in turn limits the strain the co-scheduled application can place on the interconnect and memory system, thus improving the performance of RSA. Figure 7.3 shows that RSA is particularly sensitive to being co-scheduled with SAD and SHA1.

## 7.2 Conclusion

In contrast to cooperative or pre-emptive multitasking where QoS is provided by guaranteeing a minimum amount of CPU time to an application, spatial multitasking can be used to vary the amount of SMs an application receives to provide QoS. As shown in Figure 7.1 and Table 7.1 spatial multitasking can provide a significant improvement in total system performance over cooperative multitasking while providing QoS guarantees to an application. I have investigated a simple QoS scheme here where an application is statically provisioned a fixed number of SMs to provide QoS. The remaining unassigned SMs are partitioned among applications that do not require QoS using the heuristics presented in Chapter 6. This static QoS scheme works well for some applications as shown in Table 7.2. However, a number of applications show significant performance variation depending on the applications they share the GPU with. These applications would require significant over-provisioning of SMs to provide strong QoS guarantees making this simple static QoS method inefficient.

An alternative dynamic QoS partitioning scheme using spatial multitasking can choose the number of SMs reserved for an application at runtime based on what application it is co-scheduled with. Table 7.3 shows with certain applications there is a significant benefit to this dynamic scheme over a simple static scheme. Figure 7.3 shows the sensitivity of RSA to the application it is co-scheduled with. RSA is an example of an application where total system performance benefits greatly from dynamic QoS compared to static QoS.

# 8 RECENT DEVELOPMENTS, FUTURE RESEARCH, AND CONCLUSIONS

## 8.1 Recent Developments and Future Research

Shortly after this work was completed and published [69, 72], NVIDIA announced their new Kepler GPU architecture. Kepler implements at least some form of GPGPU spatial multitasking, or as NVIDIA has named it: Hyper-Q. Details are sparse at the time of writing this document because Kepler was unveiled so recently and hardware will not be available to consumers for some time yet. However, it is clear Kepler supports spatial multitasking of up to 32 independent CUDA streams. These CUDA streams can be from multiple processes and/or multiple threads within the same process. Further details on NVIDIA's implementation of spatial multitasking remain unpublished, but an implementation of spatial multitasking in a commercial product serves as validation of the research I have conducted on this topic and the publications I have produced on the subject of spatial multitasking. NVIDIA's assessment that spatial multitasking increases GPU utilization and therefore improves total system performance agrees with my published results. My investigations of scheduling, SM migration, and QoS pave the way for future research in this area and future GPGPU products.

A number of areas related to spatial multitasking remain open for research. Notably, my research only evaluates static SM partitioning heuristics for spatial multitasking. In a real system, spatial multitasking requires dynamic partitioning heuristics. When a kernel starts or completes execution on the GPU, SMs must be redistributed among running applications. I have presented a preliminary investigation into migrating an SM from one running application and assigning it to a new application. Investigation is needed into the hardware mechanisms to support the wait, terminate, and migrate strategies I have presented. Dynamic partitioning heuristics allow SMs to be redistributed each time a kernel

starts and stops. This can lead to performance improvements over the static partitioning I have investigated because the SM partitioning can be tailored for each combination of kernels sharing the GPU not just each combination of applications sharing the GPU.

Another area that requires study is effectively combining spatial multitasking with temporal multitasking. When the GPU becomes over-committed and applications cannot be simultaneously scheduled under spatial multitasking, there is no choice but to fall back on cooperative or preemptive multitasking. Preemptive multitasking can improve average response time over cooperative multitasking but there is a large amount of GPU state associated with each process that needs to be saved and restored on each context switch.

Combining power-aware scheduling and resource partitioning could also prove beneficial. However, the most energy will be saved by reducing the amount of time and portion of the GPU that is active, so when there are several applications sharing the GPU, the best performing partitioning heuristic is also likely to be the most power-efficient.

Evaluating more advanced dynamic QoS methods for GPGPU spatial multitasking is another area open for future work. As I have shown in Chapter 7, the performance of many applications is sensitive to interference from co-scheduled applications. Varying the SM partitioning during execution allows less conservative SM reservations to be set for QoS applications; improving performance for best-effort applications. The mechanisms and metrics exposed by the GPU and system software to support dynamic QoS on spatial multitasking need to be developed.

Virtual memory should be implemented on the GPU to ensure memory isolation among applications sharing the GPU via spatial multitasking. An application should not be able to read and write into another application's memory. This can be implemented with a simple mechanism such as segmented memory or something more complex such as protected pages. Paging offers the additional benefit of swapping data to slower memory when data exceeds the capacity of the GPU main memory. GPUs are architected to tolerate long

latency events by multiplexing a large number of threads on each SM. A backing memory for swapping pages that is high-bandwidth, but still long latency to access may be well suited for the GPU.

Recent platforms have coupled CPUs and GPUs closely on the same die. By closely coupling the CPU and GPU the overhead of offloading computation on the GPU and sharing data between the CPU and GPU is reduced. This reduction in overhead enables programmers to realize performance gains from much shorter GPGPU kernels. Shorter GPGPU kernels increase the burden of scheduling GPU resources by increasing the frequency that SMs must be repartitioned among applications. Future work should investigate fast and efficient resource partitioning mechanisms and should evaluate where these scheduling decisions need to be made; in the OS, GPU driver/runtime, or GPU hardware. Closely coupled CPUs and GPUs also make it easier to execute a single OpenCL kernel in parallel on the GPU and CPU. Future work should explore spatial multitasking that spans the CPU and GPU.

In this research, my implementation of spatial multitasking explicitly partitioned GPU SMs among applications. This ensures some level of performance isolation among processes. Performance isolation can be further improved by partitioning other GPU resources as well, such as shared cache, interconnect, and memory bandwidth. Alternatively, total system performance benefits at the cost of performance isolation among applications may be realized by allowing multiple applications to share a single SM much like *simultaneous multithreading* (SMT) on CPUs.

## 8.2 Conclusions

I have proposed and evaluated spatial multitasking for GPGPU applications. Spatial multitasking allows multiple applications to simultaneously share the GPU by partitioning

its resources among applications rather than, or in addition to, time multiplexing applications, as is done by cooperative and preemptive multitasking. I presented application characterizations that indicate many GPGPU applications fail to utilize GPU resources fully. GPGPU applications are even more likely to exhibit unbalanced resource utilization in future GPUs—application performance does not typically scale linearly with increases in GPU resources such as SMs and memory and interconnect bandwidth.

I have implemented spatial multitasking in GPGPU-Sim, a cycle-accurate execution-driven GPU simulator. Simulation results indicate that spatial multitasking has the potential to offer significant performance benefits compared to cooperative multitasking by executing applications in parallel. Spatial multitasking out-performs cooperative multitasking by allowing applications to share the GPU, thus using available GPU resources more efficiently.

I have evaluated several heuristics for partitioning SMs among applications sharing the GPU via spatial multitasking. Smart Even partitioning, not much more than a simple even split of SMs among applications, showed average speedups of 1.16, 1.24, and 1.32 over cooperative multitasking for two, three, and four applications sharing the GPU, respectively. The Rounds heuristic uses information about the thread configuration of GPGPU kernels to optimize SM partitioning. The Profile heuristic uses performance information profiled in advance to distribute SMs among applications. The Fair heuristic attempts to equalize performance degradation across applications when partitioning SMs. Spatial multitasking with these heuristics show an average speedup of 1.16 to 1.17, 1.25 to 1.26, and 1.32 for two, three, and four applications sharing the GPU, respectively.

Finally, spatial multitasking provides opportunities for new, flexible scheduling and QoS techniques that can further improve the user experience. I have performed a case study of providing QoS for the SAD application. I provide QoS through the simple mechanism of dedicating a fixed number of SMs to SAD. I show improved system performance with spatial multitasking over cooperative multitasking while still providing sufficient performance

for SAD. I have also evaluated the performance sensitivity of a number of applications to co-scheduling with other applications through spatial multitasking. This evaluation indicates several applications show little performance variation based on what applications they are co-scheduled with. For these applications one can make strong guarantees on performance with little need to over-provision the number of SMs they receive under spatial multitasking. However, other applications exhibit large performance variations depending on which applications they are co-scheduled with. SMs need to be over-provisioned for these applications in order to make strong performance guarantees. In these cases, more advanced QoS methods are required to reduce the overhead of over-provisioning SMs. Finally, I show significant performance improvements could be achieved by eliminating SM over-provisioning through dynamic SM reservation based on which applications are co-scheduled together.

Together, the ideas and approaches that I have presented in this dissertation allow the GPU to be efficiently shared by multiple GPGPU applications. Previously, the usefulness of the GPU as a general-purpose accelerator was limited due to the GPGPU multitasking model. The GPU was treated as a single compute unit, with a single GPGPU kernel executing to completion before a new application could use the GPU. In this dissertation I have presented GPGPU spatial multitasking. Not only does spatial multitasking allow multiple applications to execute in parallel on the GPU, but it improves total system throughput by more efficiently using GPU resources. An improved multitasking model makes the GPU more attractive as an accelerator which will make GPGPU applications more prevalent on a variety of devices.

# A DERIVATION OF APPLICATION EXECUTION TIMES IN ISOLATION

Figures A.1 and A.2 (reproduced from Section 5.3.2) show a generalized timeline of two applications sharing the GPU using either the wait, migrate, or terminate approach to thread migration. The execution time of application A and application B is denoted as $E_A(y)$ and $E_B(y)$, respectively, where $y$ is $W1$ for wait case 1, M1 for migrate case 1, etc.

The execution time in isolation of applications A and B for wait in case 1 are:

$$
\begin{aligned}
E_A(W1) &= T_{A1} + T_W + ST_{S1} \\
E_B(W1) &= ST_{S1} + T_{B1}
\end{aligned}
$$

During $T_{A1}$ application A is running in isolation and therefore at full speed. During $T_W$ application B has started but is not running because SMs are not assigned to application B until the threads currently using them from application A complete. During $T_W$ application A runs at full speed and application B does not run at all. At the start of $T_{S1}$ the threads we waited on from application A have completed and SMs have now been assigned to application B. During $T_{S1}$ application A and application B are each using a subset of the GPU's SMs. During this time their performance is reduced by a factor of S. For spatial multitasking S is generally between 0.5 and 1.0. At the end of $T_{S1}$ application A completes GPU execution. During $T_{B1}$ application B executes in isolation at full speed.

The execution time in isolation of applications A and B for wait in case 2 are:

$$
\begin{aligned}
E_A(W2) &= T_{A1} + T_W + ST_{S4} + T_{A3} \\
E_B(W2) &= ST_{S4}
\end{aligned}
$$

During $T_{A1}$ application A is running in isolation and therefore at full speed. During $T_W$

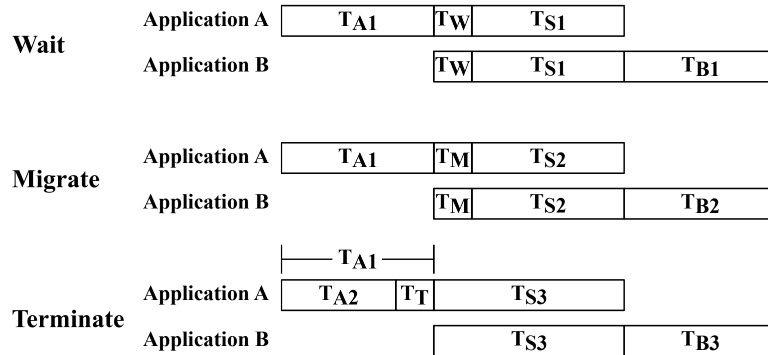



Figure A.1: Timing of the wait, migrate, and terminate approaches to thread migration. Case 1: Application A finishes before application B.

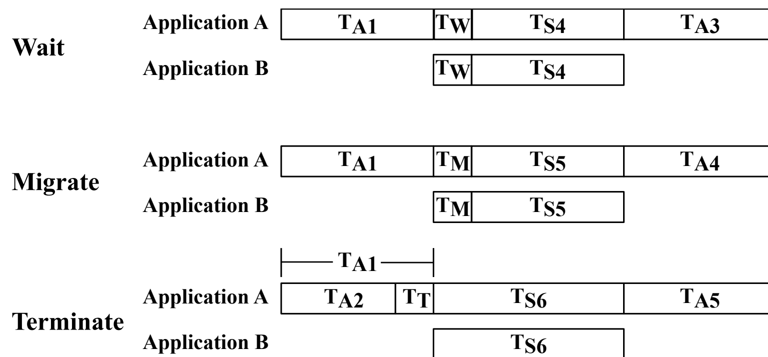



Figure A.2: Timing of the wait, migrate, and terminate approaches to thread migration. Case 2: Application A finishes after application B.

application B has started but is not running because SMs are not assigned to application B until the threads currently using them from application A complete. During $T_W$ application A runs at full speed and application B does not run at all. At the start of $T_{S4}$ the threads we waited on from application A have completed and SMs have now been assigned to application B. During $T_{S4}$ application A and application B are each using a subset of the GPU's SMs. During this time their performance is reduced by a factor of S. At the end of $T_{S4}$ application B completes GPU execution. During $T_{A3}$ application A executes in isolation at full speed.

The execution time in isolation of applications A and B for migrate in case 1 are:

$$E_A(M1) = T_{A1} + ST_M + ST_{S2}$$
$$E_B(M1) = ST_{S2} + T_{B2}$$

During $T_{A1}$ application A is running in isolation and therefore at full speed. During $T_M$ application B has started but is not running because some SMs are being preempted from application A to give to application B. During $T_M$ application A is unable use the SMs being preempted so it is running with a subset of the available SMs. This causes the performance of application A to be scaled by S. At the start of $T_{S2}$ the migration has completed and application B is able to execute. During $T_{S2}$ application A and application B are each using a subset of the GPU's SMs. During this time their performance is reduced by a factor of S. At the end of $T_{S2}$ application A completes GPU execution. During $T_{B2}$ application B executes in isolation at full speed.

The execution time in isolation of applications A and B for migrate in case 2 are:

$$E_A(M2) = T_{A1} + ST_M + ST_{S5} + T_{A4}$$
$$E_B(M2) = ST_{S5}$$

During $T_{A1}$ application A is running in isolation and therefore at full speed. During $T_M$ application B has started but is not running because some SMs are being preempted from application A to give to application B. During $T_M$ application A is unable use the SMs being preempted so it is running with a subset of the available SMs. This causes the performance of application A to be scaled by S. At the start of $T_{S5}$ the migration has completed and application B is able to execute. During $T_{S5}$ application A and application B are each using

a subset of the GPU's SMs. During this time their performance is reduced by a factor of S. At the end of $T_{S5}$ application B completes GPU execution. During $T_{A4}$ application A executes in isolation at full speed.

The execution time in isolation of applications A and B for terminate in case 1 are:

$$E_A(T1) = T_{A2} + ST_T + ST_{S3}$$
$$E_B(T1) = ST_{S3} + T_{B3}$$

During $T_{A2}$ application A is running in isolation and therefore at full speed. Application B starts at the beginning of $T_{S3}$, this causes some of application A's threads to be terminated so some SMs can be taken from application A and assigned to application B. This lost work is modeled by assuming during $T_T$ application A was executing with a subset of the GPU's SMs, and therefore the performance of application A during $T_T$ is reduced by a factor of S. During $T_{S3}$ application A and application B are each using a subset of the GPU's SMs. During this time their performance is reduced by a factor of S. At the end of $T_{S3}$ application A completes GPU execution. During $T_{B3}$ application B executes in isolation at full speed.

The execution time in isolation of applications A and B for terminate in case 2 are:

$$E_A(T2) = T_{A2} + ST_T + ST_{S6} + T_{A5}$$
$$E_B(T2) = ST_{S6}$$

During $T_{A2}$ application A is running in isolation and therefore at full speed. Application B starts at the beginning of $T_{S6}$, this causes some of application A's threads to be terminated so some SMs can be taken from application A and assigned to application B. This lost work is modeled by assuming during $T_T$ application A was executing with a subset of the GPU's

SMs, and therefore the performance of application A during $T_T$ is reduced by a factor of $S$. During $T_{S6}$ application A and application B are each using a subset of the GPU's SMs. During this time their performance is reduced by a factor of $S$. At the end of $T_{S6}$ application B completes GPU execution. During $T_{A5}$ application A executes in isolation at full speed.

# B PROOF OF WAIT IS BETTER THAN TERMINATE AND MIGRATE

The cost of wait, migrate, and terminate for case 2 are known from Section 5.3.2:

$$C_{W2} = T_{A1} + T_W + T_{S4} + T_{A3}$$
$$C_{M2} = T_{A1} + T_M + T_{S5} + T_{A4}$$
$$C_{T2} = T_{A2} + T_T + T_{S6} + T_{A5}$$

The execution time in isolation of application A and application B in terms of the times from wait, migrate, and terminate for case 2 is also known:

$$E_A(W2) = T_{A1} + T_W + ST_{S4} + T_{A3}$$
$$E_A(M2) = T_{A1} + ST_{S5} + T_{A4}$$
$$E_A(T2) = T_{A2} + ST_T + ST_{S6} + T_{A5}$$
$$E_B(W2) = ST_{S4}$$
$$E_B(M2) = ST_{S5}$$
$$E_B(T2) = ST_{S6}$$

I begin by proving $C_{W2} < C_{M2}$. We know that execution time of application A and application B remains unchanged in isolation. Therefore from application A we know:

$$E_A(W2) = E_A(M2)$$
$$T_{A1} + T_W + ST_{S4} + T_{A3} = T_{A1} + ST_{S5} + T_{A4}$$
$$T_{A4} = T_W + ST_{S4} + T_{A3} - ST_{S5} \quad \text{(B.1)}$$

And from application B we know:

$$\begin{aligned} E_B(W2) &= E_B(M2) \\ ST_{S4} &= ST_{S5} \\ T_{S4} &= T_{S5} \end{aligned} \tag{B.2}$$

Solving the inequality:

$$C_{W2} < C_{M2}$$
$$T_{A1} + T_W + T_{S4} + T_{A3} < T_{A1} + T_M + T_{S5} + T_{A4}$$
$$T_W + T_{S4} + T_{A3} < T_M + T_{S5} + T_{A4}$$

Substituting for $T_{A4}$ from Equation B.1:

$$T_W + T_{S4} + T_{A3} < T_M + T_{S5} + T_W + ST_{S4} + T_{A3} - ST_{S5}$$
$$(1-S)T_{S4} < T_M + (1-S)T_{S5}$$

Substituting for $T_{S4}$ from Equation B.2:

$$(1-S)T_{S5} < T_M + (1-S)T_{S5}$$
$$T_M > 0$$

I assume that $T_M$ (the time it takes to migrate threads to a new SM), is some positive non-zero time. Therefore $C_{W2} < C_{M2}$ and we should always choose wait over migrate for case 2.

Next I prove $C_{W2} < C_{M2}$. From the execution time in isolation of application A:

$$E_A(W2) = E_A(T2)$$
$$T_{A1} + T_W + ST_{S4} + T_{A3} = T_{A2} + ST_T + ST_{S6} + T_{A5}$$
$$T_{A5} = T_{A1} + T_W + ST_{S4} + T_{A3} - T_{A2} - ST_T - ST_{S6} \tag{B.3}$$

And from application B:

$$E_B(W2) = E_B(M2)$$
$$ST_{S4} = ST_{S6}$$
$$T_{S4} = T_{S6} \tag{B.4}$$

Solving the inequality:

$$C_{W2} < C_{T2}$$
$$T_{A1} + T_W + T_{S4} + T_{A3} < T_{A2} + T_T + T_{S6} + T_{A5}$$

Substituting for $T_{A5}$ from Equation B.3:

$$T_{A1} + T_W + T_{S4} + T_{A3} < T_{A2} + T_T + T_{S6} + T_{A1} + T_W + ST_{S4} + T_{A3} - T_{A2} - ST_T - ST_{S6}$$
$$(1 - S)T_{S4} < (1 - S)T_T + (1 - S)T_{S6}$$

Substituting for $T_{S4}$ from Equation B.4:

$$(1-S)T_{S6} < (1-S)T_T + (1-S)T_{S6}$$
$$(1-S)T_T > 0$$

If we assume $0 < S < 1$ then:

$$T_T > 0$$

I also assume that $T_T$ (the execution time lost due to terminating threads), is some positive non-zero time. Therefore when $0 < S < 1$ then $C_{W2} < C_{T2}$ and we should always choose wait over migrate for case 2. When $S = 1$ this means there is no change in performance when application share the GPU through spatial multitasking versus running in isolation; then the cost of wait and terminate in case 2 is the same. In the unlikely event that $S > 1$ this means applications run faster when sharing the GPU than in isolation; then it would actually be better to choose terminate over wait in case 2.

# REFERENCES


[1] A. L. Shimpi, "The iPhone 3GS hardware exposed & analyzed," June 2009. [Online]. Available: http://www.anandtech.com/show/2782/3

[2] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," in *Proceedings of the IEEE*, vol. 96, no. 3, March 2008, pp. 879–899.

[3] S.-T. Yang, T.-K. Lin, and S.-Y. Chien, "Real-time motion estimation for 1080p videos on graphics processing units with shared memory optimization," in *IEEE Workshop on Signal Processing Systems*, October 2009, pp. 297–302.

[4] "NVIDIA's next generation CUDA compute architecture: Fermi," NVIDIA Corporation, 2009. [Online]. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[5] S. Vaughn-Nichols, "Vendors draw up a new graphics-hardware approach," *Computer*, vol. 42, no. 5, pp. 11–13, May 2009.

[6] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," in *Eurographics 2005, State of the Art Reports*, August 2005, pp. 21–51.

[7] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.

[8] "AMD "Close To Metal" technology unleashes the power of stream computing," November 2006. [Online]. Available: http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543~114147,00.html

[9] "NVIDIA unveils CUDA - The GPU computing revolution begins," November 2006. [Online]. Available: http://www.nvidia.com/object/IO_37226.html

[10] T. R. Halfhill, "Parallel processing with CUDA," Microprocessor Report, January 2008. [Online]. Available: www.nvidia.com/docs/IO/55972/220401_Reprint.pdf

[11] "NVIDIA GeForce GTX 200 GPU datasheet." [Online]. Available: http://www.nvidia.com/docs/IO/55506/GPU_Datasheet.pdf

[12] D. Kanter, "NVIDIA's GT200: Inside a parallel processor," September 2008. [Online]. Available: http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242

[13] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," in *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007, pp. 407–420.

[14] A. Bakhoda, G. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *ISPASS '09: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, April 2009.

[15] S. Pronovost, H. Moreton, and T. Kelley, "Windows display driver model (WDDM) v2 and beyond," WinHEC '06. [Online]. Available: http://download.microsoft.com/download/5/b/9/5b97017b-e28a-4bae-ba48-174cf47d23cd/PRI103_WH06.ppt

[16] "NVIDIA CUDA FAQ ver. 2.1," December 2008. [Online]. Available: http://forums.nvidia.com/index.php?showtopic=84440

[17] *Timeout Detection and Recovery of GPUs through WDDM*, Microsoft Corporation, April 2009. [Online]. Available: http://www.microsoft.com/whdc/device/display/wddm_timeout.mspx

[18] *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, Rev. 3.17 ed., AMD, June 2010. [Online]. Available: http://support.amd.com/us/Processor_TechDocs/24593.pdf

[19] J. Lengyel, M. Reichert, B. R. Donald, and D. P. Greenberg, "Real-time robot motion planning using rasterizing computer graphics hardware," *SIGGRAPH '90: Proceedings of the ACM SIGGRAPH*, vol. 24, no. 4, pp. 327–335, 1990.

[20] C. J. Thompson, S. Hahn, and M. Oskin, "Using modern graphics architectures for general-purpose computing: A framework and analysis," in *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, 2002, pp. 306–317.

[21] R. Wu, B. Zhang, and M. Hsu, "Clustering billions of data points using GPUs," in *UCHPC-MAW '09: Proceedings of the Combined Workshops on UnConventional High Performance Computing Workshop Plus Memory Access Workshop*, 2009, pp. 1–6.

[22] J. M. Nageswaran, N. Dutt, J. L. Krichmar, A. Nicolau, and A. Veidenbaum, "Efficient simulation of large-scale spiking neural networks using CUDA graphics processors," in *IJCNN '09: International Joint Conference on Neural Networks*, 2009.

[23] M. Boyer, D. Tarjan, S. T. Acton, and K. Skadron, "Accelerating leukocyte tracking using CUDA: A case study in leveraging manycore coprocessors," in *IPDPS '09: Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, May 2009.

[24] T. Preis, P. Virnau, W. Paul, and J. J. Schneider, "GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model," *Journal of Computational Physics*, vol. 228, no. 12, pp. 4468–4477, 2009.

[25] T. Reichl, J. Passenger, O. Acosta, and O. Salvado, "Ultrasound goes GPU: Real-time simulation using CUDA," in *Medical Imaging 2009: Visualization, Image-Guided Procedures, and Modeling*, 2009.

[26] Z. Feng and P. Li, “Multigrid on GPU: Tackling power grid analysis on parallel SIMT platforms,” in *ICCAD ’08: Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, 2008, pp. 647–654.

[27] Maxime, “Ray tracing.” [Online]. Available: http://www.nvidia.com/cuda

[28] S. A. Manavski, “CUDA compatible GPU as an efficient hardware accelerator for AES cryptography,” in *ICSPC ’07: IEEE International Conference on Signal Processing and Communications*, November 2007, pp. 65–68.

[29] A. K. Anton Obukhov, *Discrete Cosine Transform for 8x8 Blocks with CUDA*, v1.0 ed., NVIDIA Corporation, October 2008. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/2_2/sdk/website/projects/dct8x8/doc/dct8x8.pdf

[30] S. Brodhead, “GPU flame fractal renderer.” [Online]. Available: http://sourceforge.net/projects/flam4/

[31] W. J. van der Laan, “GPU-accelerated Dirac video codec.” [Online]. Available: http://diracvideo.org

[32] J. W. Sheaffer, D. Luebke, and K. Skadron, “A flexible simulation framework for graphics architectures,” in *HWWS ’04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 2004, pp. 85–94.

[33] J. W. Sheaffer, K. Skadron, and D. P. Luebke, “Studying thermal management for graphics-processor architectures,” in *ISPASS ’05: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2005, pp. 54–65.

[34] V. del Barrio, C. Gonzalez, J. Roca, A. Fernandez, and R. Espasa, “ATTILA: A cycle-level execution-driven simulator for modern GPU architectures,” in *ISPASS ’06: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, March 2006, pp. 231–241.

[35] S. Collange, D. Defour, and D. Parello, “Barra, a modular functional GPU simulator for GPGPU,” UniversitÃ© de Perpignan, Tech. Rep. v2, February 2009. [Online]. Available: http://hal.archives-ouvertes.fr/hal-00359342/en/

[36] K. Rupnow, J. Adriaens, W. Fu, and K. Compton, “Accurately evaluating application performance in simulated hybrid multi-tasking systems,” in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, ser. FPGA ’10. New York, NY, USA: ACM, 2010, pp. 135–144.

[37] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, “Introduction to the Cell multiprocessor,” *IBM Journal of Research and Development*, vol. 49, no. 4/5, pp. 589–604, 2005.

[38] *Cell Broadband Engine Programming Handbook*, Ver. 1.11 ed., IBM, May 2008. [Online]. Available: https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/1741C509C5F64B3300257460006FD68D/$file/CellBE_PXCell_Handbook_v1.11_12May08_pub.pdf

[39] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: A many-core x86 architecture for visual computing," in *SIGGRAPH '08: ACM SIGGRAPH 2008 Papers*, 2008, pp. 1–15.

[40] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang, "EXOCHI: Architecture and programming environment for a heterogeneous multi-core multithreaded system," in *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007, pp. 156–166.

[41] W. Fu and K. Compton, "Active kernel monitoring to combat scheduler gaming in reconfigurable computing systems," in *FPL '08: Proceedings of the International Conference on Field Programmable Logic and Applications*, 2008, pp. 611–614.

[42] C. McCann, R. Vaswani, and J. Zahorjan, "A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors," *ACM Transactions on Computer Systems*, vol. 11, no. 2, pp. 146–178, 1993.

[43] J. K. Ousterhout, "Scheduling techniques for concurrent systems," in *Proceedings of the 3rd International Conference on Distributed Computing Systems*, 1982, pp. 22–30.

[44] T. Blank, "The MasPar MP-1 architecture," in *Proceedings of IEEE Compcon=*, 1990, pp. 20–24.

[45] W. D. Hillis, *The Connection Machine*. MIT Press, 1986.

[46] C. Du, X.-H. Sun, and M. Wu, "Dynamic scheduling with process migration," in *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, 2007, pp. 92–99.

[47] J. A. Stratton, S. S. Stone, and W.-M. W. Hwu, "MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs," in *LCPC '08: Proceedings of the 21st International Workshop on Languages and Compilers for Parallel Computing*, 2008, pp. 16–30.

[48] G. Diamos, A. Kerr, and M. Kesavan, "Translating GPU binaries to tiered SIMD architectures with Ocelot," Georgia Institute of Technology, Tech. Rep., 2009. [Online]. Available: http://www.cercs.gatech.edu/tech-reports/tr2009/abstracts/01.html

[49] "GPUOcelot: A binary translator framework for PTX." [Online]. Available: http://code.google.com/p/gpuocelot/

[50] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: A compiler framework for automatic translation and optimization," in *PPoPP '09: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2009, pp. 101–110.

[51] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W.-M. W. Hwu, "High-performance CUDA kernel execution on FPGAs," in *ICS '09: Proceedings of the 23rd International Conference on Supercomputing*, 2009, pp. 515–516.

[52] M. D. McCool, "Data-parallel programming on the Cell BE and the GPU using the RapidMind development platform," in *GSPx Multi-core Applications Conference*, November 2006.

[53] *The OpenCL Specification*, v1.0 ed., Khronos OpenCL Working Group, May 2009. [Online]. Available: http://www.khronos.org/opencl/

[54] V. J. Jiménez, L. Vilanova, I. Gelado, M. Gil, G. Fursin, and N. Navarro, "Predictive runtime code scheduling for heterogeneous architectures," in *HiPEAC '09: Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, 2009, pp. 19–33.

[55] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "TimeGraph: GPU scheduling for real-time multi-tasking environments," in *USENIXATC'11: Proceedings of the 2011 USENIX conference on USENIX annual technical conference*. Berkeley, CA, USA: USENIX Association, 2011.

[56] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, "PTask: Operating system abstractions to manage GPUs as compute devices," in *SOSP '11: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. New York, NY, USA: ACM, 2011, pp. 233–248.

[57] G. A. Elliott and J. H. Anderson, "Globally scheduled real-time multiprocessor systems with gpus," *Real-Time Syst.*, vol. 48, no. 1, pp. 34–74, Jan. 2012.

[58] "CUDA zone – The resource for CUDA developers." [Online]. Available: http://www.nvidia.com/object/cuda_home.html

[59] "GPGPU benchmarks." [Online]. Available: http://ercbench.ece.wisc.edu

[60] R. Szerwinski and T. Güneysu, "Exploiting the power of GPUs for asymmetric cryptography," in *CHES '08: Proceedings of the 10th International Workshop on Cryptographic Hardware and Embedded Systems*, 2008, pp. 79–99.

[61] "StoreGPU." [Online]. Available: http://www.ece.ubc.ca/~samera/projects/StoreGPU/code/

[62] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, and M. Ripeanu, “StoreGPU: Exploiting graphics processing units to accelerate distributed storage systems,” in *HPDC '08: Proceedings of the 17th International Symposium on High Performance Distributed Computing*, 2008, pp. 165–174.

[63] A. Kharlamov and V. Podlozhnyuk, “Image denoising,” NVIDIA Corporation, June 2007. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html#imageDenoising

[64] “CUDPP: CUDA data parallel primitives library.” [Online]. Available: http://gpgpu.org/developer/cudpp

[65] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, “Scan primitives for GPU computing,” in *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, 2007, pp. 97–106.

[66] “Parboil benchmark suite.” [Online]. Available: http://impact.crhc.illinois.edu/parboil.php

[67] “GPGPU-Sim.” [Online]. Available: http://www.gpgpu-sim.org

[68] “NVIDIA Quadro FX 5800.” [Online]. Available: http://www.nvidia.com/object/product_quadro_fx_5800_us.html

[69] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte, “The case for GPGPU spatial multitasking,” in *HPCA '12: Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, 2012.

[70] M. Herlihy and J. E. B. Moss, “Transactional memory: architectural support for lock-free data structures,” in *ISCA '93: Proceedings of the 20th annual international symposium on computer architecture*, 1993, pp. 289–300.

[71] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood, “LogTM: Log-based transactional memory,” in *HPCA '06: In Proceedings of the Twelfth IEEE Symposium on High-Performance Computer Architecture*, 2006.

[72] J. T. Adriaens, P. Aguilera, K. Compton, N. S. Kim, and M. J. Schulte, “The case for GPGPU spatial multitasking,” in *SRC TECHCON '11*, 2011.