The Case for GPGPU Spatial Multitasking

Jacob T. Adriaens, Katherine Compton, Nam Sung Kim
Department of Electrical and Computer Engineering
University of Wisconsin - Madison
jtadriaens@wisc.edu, kati@engr.wisc.edu, nskim3@wisc.edu

Michael J. Schulte
AMD Research
Michael.Schulte@amd.com

Abstract

The set-top and portable device market continues to grow, as does the demand for more performance under increasing cost, power, and thermal constraints. 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. In this paper, we propose and make the case for a 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. We demonstrate the potential benefits of spatial multitasking with an analysis and characterization of General-Purpose GPU (GPGPU) applications. We find 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. We then implement spatial multitasking and compare it to cooperative multitasking using simulation. We evaluate several heuristics for partitioning GPU stream multiprocessors (SMs) among applications and find spatial multitasking shows an average speedup of up to 1.19 over cooperative multitasking when two applications are sharing the GPU. Speedups are even higher when more than two applications are sharing the GPU.

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 4, for example, contains a programmable GPU in addition to a general-purpose CPU and several *Application-Specific Instruction Processors* (ASIPs). Transforming the GPU from a graphics and compute offload device to a general-purpose data-parallel processor has the potential to enable entirely new classes of applications that were previously unavailable on 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 [22]. A range of applications, from scientific computing to multimedia, are well-suited to this form of parallelism and achieve large speedups on a GPU. 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 [31].

Unfortunately, GPUs have very primitive support for multitasking, a key feature of modern computing systems. Multitasking provides concurrent execution of multiple applications on a single device. Advanced multitasking is critical for preserving user responsiveness and satisfying *quality-of-service* (QoS) requirements. NVIDIA's Fermi supports co-executing multiple tasks from the *same* application on a single GPU [19]. However, even Fermi does not allow multiple *different* GPGPU applications to access GPU resources simultaneously. Other applications needing the GPU must wait until the application occupying the GPU voluntarily yields control. Having the application voluntarily yield control of the GPU is a form of *cooperative multitasking*. In contrast, on the CPU, the operating system (OS) typically uses *preemptive multitasking*—suspending and later resuming applications to time-share the CPU without the applications' intervention or control. Both cooperative and preemptive multitasking are forms of temporal multitasking. Finally, multi-core CPUs support spatial multitasking, which allows multiple applications to execute simultaneously on different cores.

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 [30], the importance of advancing the GPU from a graphics-only co-processor to a multitasking parallel accelerator will grow. This will require development of new GPU multitasking techniques, both temporal and spatial.

In this paper, we present a characterization of GPGPU applications for the portable and set-top markets. With this characterization, we observe GPGPU applications exhibit unbalanced GPU resource utilization. Using simulation, we then demonstrate significant performance improvements when using spatial multitasking instead of cooperative multitasking due to more efficient use of GPU resources. We also evaluate several heuristics for partitioning GPU SMs among applications sharing the GPU. The key contributions of this work are: (1) Our proposal for GPGPU spatial multitasking, which allows applications to execute simultaneously with GPU resources partitioned among them, rather than executing serially on all GPU resources. (2) A detailed characterization of GPGPU applications demonstrating many GPGPU workloads show unbalanced usage of GPU resources. (3) An evaluation of GPGPU spatial multitasking versus cooperative multitasking through cycle-accurate simulation. (4) A comparison of heuristics for partitioning SMs among applications sharing a GPU via spatial multitasking.

This paper is organized as follows. Section 2 discusses temporal multitasking, and the details of and motivation for spatial multitasking. Section 3 presents our GPGPU workload analysis, which focuses on the inefficient use of resources by applications executing in isolation on the GPU, followed by the evaluation of spatial multitasking compared to cooperative multitasking and a comparison of several SM partitioning heuristics. Section 4 presents potential hardware and software challenges faced when implementing spatial multitasking. Section 5 discusses related work, and Section 6 provides our conclusions and a discussion of our planned future work.

2. GPU Multitasking

Initially, multiple graphics applications could only share a GPU via *cooperative multitasking*, requiring applications 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 [17,24]. GPGPU applications continue to use cooperative multitasking.

In cooperative multitasking, a computation block offloaded 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 GPU computation [16]. 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. 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 GPU core, or 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 [11].

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 [2]; this and other state 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 CPU context switch.

To address the problems and challenges associated with temporal multitasking on the GPU, we 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 1 illustrates the differences among cooperative, preemptive, and spatial multitasking.

We have observed that many GPGPU workloads are tuned for a particular GPU generation and subsequent, more aggressive GPUs frequently show unbalanced resource usage by software just one generation old. Spatial multitasking can increase GPU utilization and improve system performance compared to temporal multitasking. Dividing resources among applications also reduces context switches, further improving performance compared to preemptive multitasking.

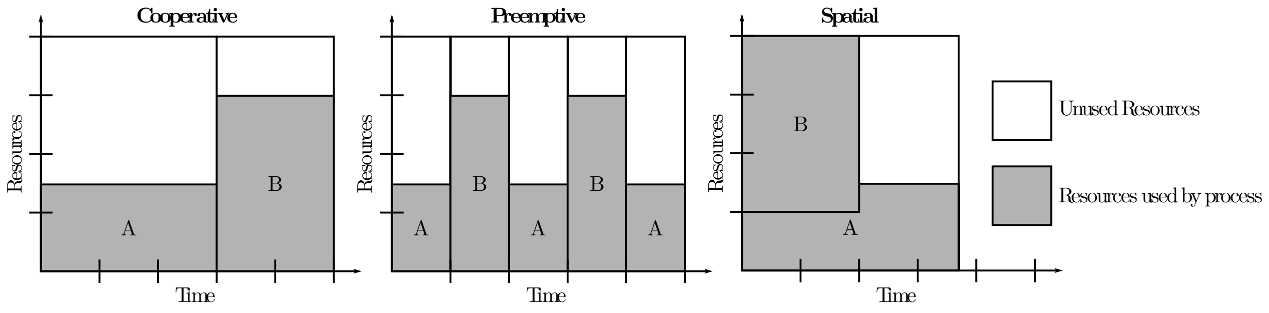

Figure 1. Illustration of multitasking methods. A and B represent applications using the GPU. The Y-axis represents the GPU resources and the X-axis represents the duration those resources are used. In spatial multitasking, the applications execute simultaneously with resources split among them.

3. Evaluation

To determine how efficiently applications use GPU resources and to evaluate potential benefits of spatial multitasking, we profile the effects of varying available GPU resources on the applications described in Section 3.1. We also simulate these applications sharing the GPU using both spatial and cooperative multitasking. Our methodology is described in Section 3.2 and the profiling results and their relevance to spatial multitasking are discussed in Section 3.3. Our evaluation of spatial multitasking is presented in Section 3.4 and Section 3.5 compares several heuristics for partitioning GPU SMs among applications sharing the GPU via spatial multitasking.

3.1. Applications

We characterize workloads targeting the portable and set-top markets. These workloads are selected from source code available online [5]. Although the workloads and input data sets target set-top and portable devices, the results should also apply to desktop devices. We have placed all workloads and input data online for public use [7]. Some workloads include CPU computation as detailed in the following workload descriptions; however, we profile only the GPU portions of computation because this is the portion directly impacted by GPU spatial multitasking.

Ray Tracing [14] 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 [13] 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 encrypts 4MB of randomly generated data.

RSA [28] performs RSA asymmetric encryption. As with AES, a single GPU kernel performs the entire encryption. The CPU performs initialization and cleanup. We use a 1,024-bit key to encrypt in parallel 128 randomly generated 128-byte messages, for a total encryption size of 16KB.

SHA1 [1, 27], 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, using SHA1, we block-hash 4MB of randomly generated data.

JPEG Encoding and Decoding [20] are partial implementations of JPEG encoding and decoding. We 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 3,072x2,304-pixel bitmap, and the quantization input is the DCT output. The decoding input is the encoding output, which we pre-generate.

Fractals [4] 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 of GPU kernels that perform the rendering in several 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 [12] 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. Our input data is a 3,072x2,304-pixel bitmap.

Radix Sort [6,26] is a single GPU kernel from the *CUDA Data Parallel Primitives* (CUDPP) library. It performs a radix sort of 2^{18} randomly generated integers.

Table 1. GPU kernel configuration. Average threads and blocks are calculated according to Equation 1.

Application	Kernels	Average Thread Blocks	Average Threads per Block
AES Decrypt	1	4,097	256
AES Encrypt	1	4,097	256
DVC	14	112	216
Fractal Generation	1	512	64
Image Denoising	1	110,592	64
JPEG Decode	1	13,824	64
JPEG Encode	2	73,694	64
RSA	1	4	32
Radix Sort	4	357	250
Ray Tracing	1	2,048	128
SAD	3	4,354	66
SHA1	1	65	64

DVC [29] 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 DVC and H.264 is that DVC 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 [23] 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. Table 1 lists the average number of thread blocks and threads per block in each application. Average thread blocks and threads per block are calculated as the weighted arithmetic mean (Equation 1) of thread 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 thread blocks or threads per block in kernel i.

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

In CUDA, a thread block cannot be split across multiple SMs, so applications with fewer thread blocks are less likely to fully utilize available SMs. For example, RSA has four thread blocks and 32 threads per block; this means RSA never utilizes more than four SMs. 32 threads per block is also relatively small, so RSA is unlikely to efficiently use even four SMs. Kernels with a small number of threads per block require more thread blocks per SM to hide memory access latency and more fully utilize the SM.

3.2. Methodology

We simulate CUDA applications with GPGPU-Sim, a cycle-accurate execution-driven GPU simulator capable of simulating CUDA and OpenCL applications [3,8]. GPGPU-Sim has been shown to correlate well with existing GPU hardware [3]. GPGPU-Sim emulates NVIDIA's virtual GPU instruction set architecture called *Parallel Thread eXecution* (PTX). GPU portions of simulated applications are emulated 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. We use the default GPGPU-Sim parameters that approximate the NVIDIA Quadro FX 5800 GPU [8, 18], shown in Table 2. Although our infrastructure models an NVIDIA GPU, these research results should apply to other GPU architectures.

As an initial step in evaluating spatial multitasking, we profiled the applications from Section 3.1 to see how they responded to different amounts of available GPU resources. Specifically, we 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. An explanation for why we simulated for 5M cycles is given in Section 3.4. The NVIDIA Quadro FX 5800 baseline values for these parameters are shown in Table 2. If we use spatial multitasking to partition SMs among applications sharing the GPU, then varying these parameters gives some insight into predicted application performance when sharing the GPU. 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 modelled when varying memory bandwidth or the number of SMs in isolation. For these reasons, we implement spatial and cooperative multitasking in GPGPU-Sim and evaluate them through simulation in Section 3.4.

Our implementation of spatial multitasking in GPGPU-Sim partitions SMs among applications with a user defined algorithm. In this implementation SMs can only be parti-

Table 2. 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

tioned at the beginning of simulation. We are able to do this because in our simulations we start all applications simultaneously. In practice, because applications do not typically start and end at the same time, spatial multitasking requires the ability to dynamically take SMs from running applications and reassign these SMs to other applications. Hardware and software mechanisms for supporting SM reassignment are the subject of future work. The GPGPU-Sim interconnect and memory model has not been modified for spatial multitasking. SMs share interconnect and memory bandwidth as if they were executing threads from a single process. GPU global and texture memory are not virtualized for this study. Applications allocate memory from the same physical pool but memory isolation is not enforced. In practice memory isolation should be provided to ensure faulty or malicious applications do not have access to another applications memory space. No modifications of GPGPU-Sim were necessary to simulate cooperative multitasking. The source code for our modified version of GPGPU-Sim is available at: http://mesa.ece.wisc.edu/gpgpu.

3.3. Application Characterization

Figures 2 to 4 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 under-utilizes the examined resource either due to a bottleneck in the system or lack of parallelism in the application. In each figure, constant parameters are set according to Table 2.

In Figure 2, we varied the number of SMs in the simulated GPU. 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

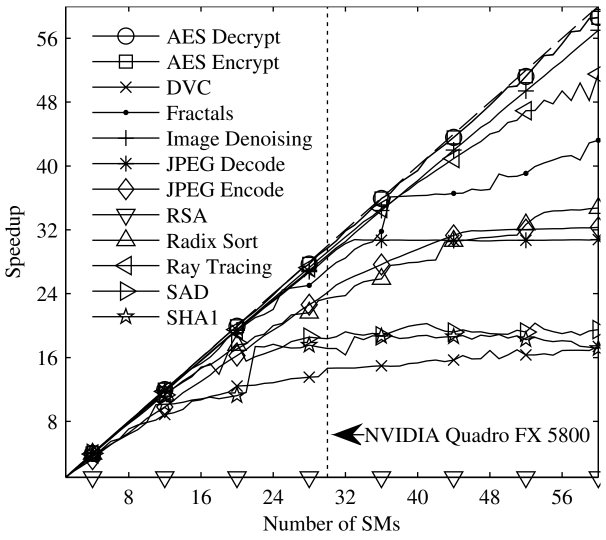

Figure 2. Speedup of GPU computation versus number of SMs. Results are normalized to one SM.

our 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 2 shows that many applications scale sub-linearly up to 30 SMs (the number in our baseline configuration), and scale even worse at expected future resource levels. As expected from the thread configuration shown in Table 1, RSA showed almost no performance change as SMs were increased.

Figure 3 examines memory frequency to study the effect of memory bandwidth limits on GPGPU applications; we use memory frequency as an approximation for bandwidth since we cannot easily model fixed-frequency bandwidth caps. 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 under-utilized 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 little demand.

In Figure 4, 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 we observed 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.

Figures 2 to 4 confirm that many GPGPU applications exhibit unbalanced GPU resource utilization. It is thus

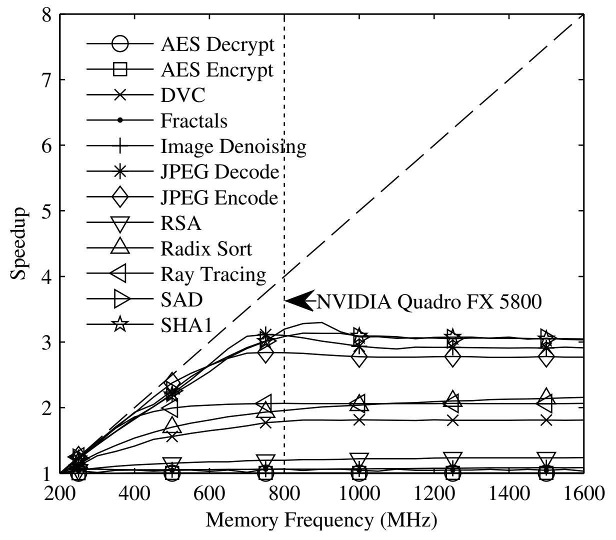

Figure 3. Speedup of GPU computation versus GPU memory frequency. Results are normalized to a 200MHz GPU memory clock. The dashed line represents linear speedup.

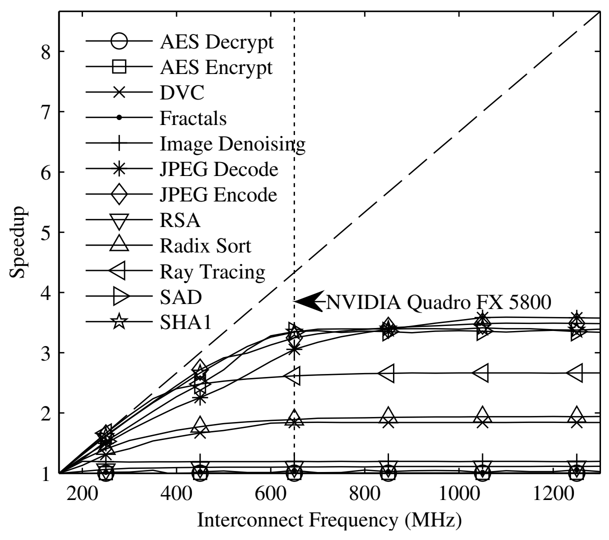

Figure 4. Speedup of GPU computation versus GPU interconnect frequency. Results are normalized to a 150MHz GPU interconnect clock. The dashed line represents linear speedup.

likely that spatial multitasking will improve system performance by allowing multiple GPGPU applications to share the GPU without significantly impacting individual performance. 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 it would 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.

We have classified each application based on the information provided in Table 1 and Figures 2 to 4. We verify our assertions regarding the impact of spatial multitasking on these classes of applications using the methodology presented in Section 3.4.

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 multitasking is implemented such that SMs are partitioned among applications sharing the GPU, these applications are unlikely to contribute significantly to increases in total system performance using spatial multitasking. These applications are likely to take roughly N times longer to execute if they are given N times fewer 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 sufficiently parallel 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 if their limiting resources do not conflict with other applications sharing the GPU. For example, two memory-bound applications sharing the GPU is not likely to improve performance, but a memory-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 do not have 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 in which SMs are partitioned among applications sharing the GPU. DVC, Radix Sort, RSA, and SHA1 are problem size-bound.

3.4. Multitasking Evaluation

In the previous section, we analyzed GPGPU applications in isolation as the resources available to them are varied. This data indicates that many applications show unbalanced GPU resource usage for current hardware and likely more for future hardware, making spatial multitasking an attractive solution. In this section, using simulation, we confirm spatial multitasking is able to out-perform cooperative multitasking. *Although we do not model preemptive multitasking for this research, in these experiments preemptive multitasking will always perform worse than cooperative multitasking due to context switch overhead.*

We have implemented both cooperative and spatial multitasking in GPGPU-Sim. Using the NVIDIA Quadro FX 5800 baseline configuration, we compare the total run time of two, three, and four applications sharing the GPU under

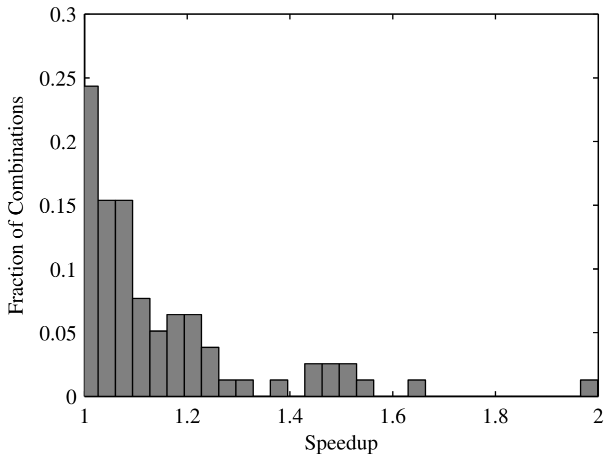

Figure 5. Distribution of speedup of spatial multitasking relative to cooperative multitasking for all combinations of two applications sharing the GPU.

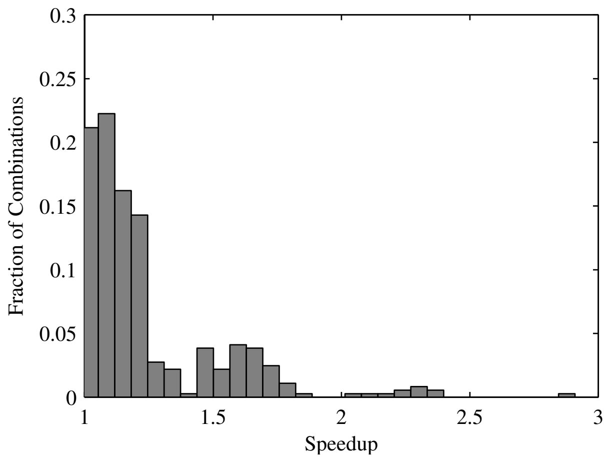

Figure 6. Distribution of speedup of spatial multitasking relative to cooperative multitasking for all combinations of three applications sharing the GPU.

spatial and cooperative multitasking. We 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 are AA, AB, AC, BB, BC, and CC. For this evaluation, kernel initialization and cleanup is not modelled for either multitasking method. This favors neither cooperative nor spatial multitasking because they both require this overhead. In this initial evaluation of spatial multitasking, SMs are naively partitioned evenly among each application. When the number of SMs is not divisible by the number of applications, the assignment is made arbitrarily with a near-even split. We have determined through experimentation that this arbitrary assignment does not significantly affect results. More intelligent methods for SM partitioning are examined in Section 3.5. GPU global memory is not virtualized for this study. Applications allocate memory from the same physical pool

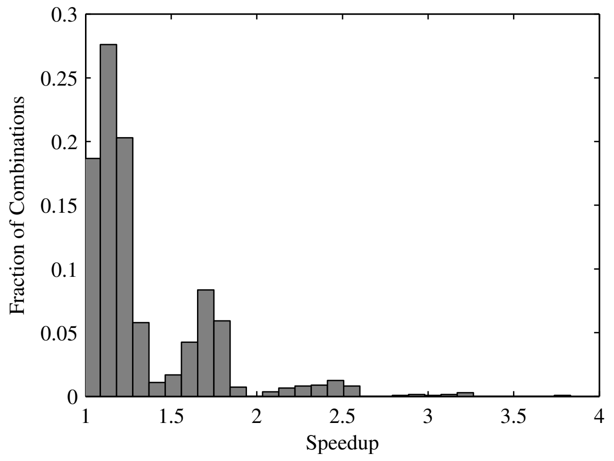

Figure 7. Distribution of speedup of spatial multitasking relative to cooperative multitasking for all combinations of four applications sharing the GPU.

but memory isolation is not enforced. In a real implementation of spatial multitasking, memory isolation should be provided to ensure faulty or malicious applications do not have access to another application's memory space.

For many applications, it is not feasible to simulate the entire execution in a reasonable amount of time since cycle-accurate GPU simulation is considerably more complex and time consuming than cycle-accurate CPU simulation due to the large number of threads that execute simultaneously in GPUs. Instead, we simulate spatial multitasking for a fixed number of cycles and measure the work completed (instructions executed). When we began experimentation, we tried a number of simulation lengths and compared how the results changed as the length of simulation changed. We found that results from shorter simulations (1M cycles or less) varied considerably, but longer simulations (greater than 1M cycles) converged on similar results for the applications evaluated in this research. This is probably due to startup variations that are not representative of the steady-state behavior of applications. We simulate for 5M GPU cycles because it is sufficiently greater than 1M cycles so that startup variations have minimal impact on performance results and is sufficiently small that simulation time is still reasonable.

We simulate each combination of applications in spatial multitasking for 5M GPU cycles; if an application completes in less than 5M cycles, it is restarted and simulation continues to ensure we observe 5M GPU cycles. We simulate each combination of applications in cooperative multitasking for a fixed number of instructions. The number of instructions for which each application is run in cooperative multitasking corresponds to the number of instructions that are completed by the application in the corresponding spatial multitasking simulation. This ensures we are comparing the same amount of work for each application between spatial and cooperative multitasking.

Table 3. Speedup of spatial multitasking compared to cooperative multitasking. SMs are naively split evenly among applications for spatial multitasking in this experiment.

	2 Apps	3 Apps	4 Apps
Geometric Mean	1.14	1.22	1.30
Maximum	2.00	2.91	3.83
Minimum	0.99	0.99	0.99

Table 4. Speedup of spatial multitasking compared to cooperative multitasking for several benchmark combinations.

	Oracle Best		Even	
Applications	SMs	Speedup	SMs	Speedup
DVC	8	1.23	15	1.14
SHA1	22		15	
AES Encrypt	25	1.01	15	1.01
Image Denoising	5		15	
JPEG Decode	20	1.08	15	1.06
Radix Sort	10		15	
JPEG Encode	25	1.08	15	1.05
SAD	5		15	

Figures 5 to 7 plot the distribution of speedup of spatial multitasking relative to cooperative multitasking for all combinations of two, three, and four applications sharing the GPU when SMs are split evenly among the applications. For the applications we have evaluated, in 75% of simulations spatial multitasking shows speedups relative to cooperative multitasking 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 3 presents the speedup of spatial multitasking compared to cooperative multitasking for all combinations of two to four applications sharing the GPU. For these results, only an even-split of SMs among applications in spatial multitasking is presented. When four applications share a GPU with 30 SMs, two applications are arbitrarily assigned eight SMs while the other two applications are assigned seven.

As shown in Table 3, 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 4 details the performance of several specific combinations of applications. We examine both Oracle Best and Even SM partitioning. Oracle Best partitioning is an exhaustive search of all possible partitions of SMs among applications that selects the partitioning leading to the greatest speedup over cooperative multitasking. Oracle is not practical to implement in a real system. 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 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. Finally, we see that pairing interconnect/memory-bound applications using spatial multitasking, in this case JPEG Encode and SAD, also outperforms cooperative multitasking. These results concur with our application classification given in Section 3.3.

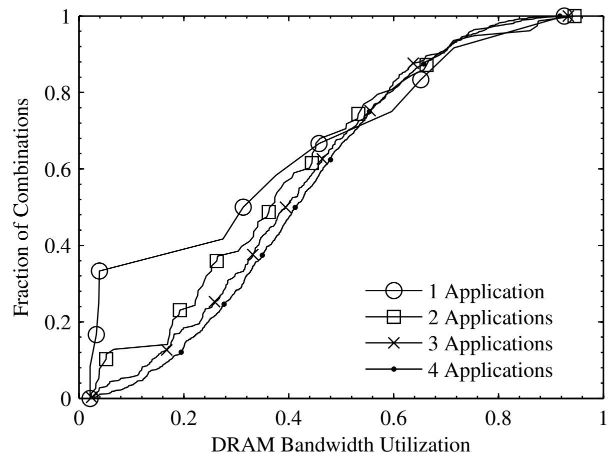

Figure 8. Cumulative distribution function of the DRAM bandwidth utilization for all combinations of applications using spatial multitasking. The Y-axis represents the fraction of combinations that have a DRAM bandwidth utilization *less than or equal to* **the corresponding point on the X-axis.**

Table 5. Summary of DRAM bandwidth utilization for all combinations of applications.

	1 App	2 Apps	3 Apps	4 Apps
Geometric 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%

Figures 8 and 9 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 and 6 summarize the results shown in Figures 8 and 9, respectively. The trend these figures and tables show is that as the number of applications sharing the GPU under spatial multitasking increases

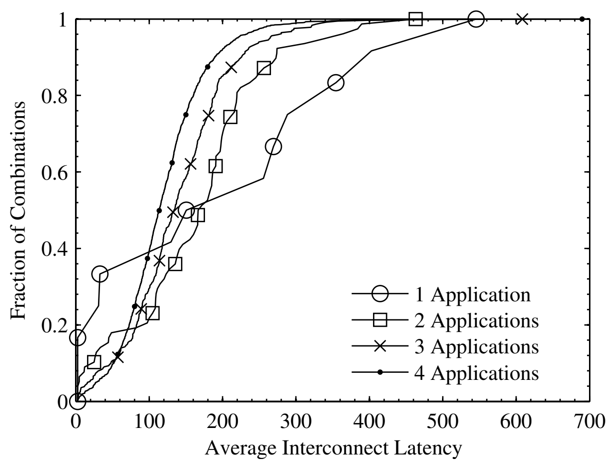

Figure 9. Cumulative distribution function of the average interconnect latency for all combinations of applications using spatial multitasking.

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 results in a decrease in average interconnect latency. We have observed for many applications when running in isolation, a 2x reduction in SMs results in more than a 2x reduction in average interconnect latency. This contributes to the reduced average interconnect latency observed in spatial multitasking compared to cooperative multitasking.

3.5. SM Partitioning

In the previous section we analyzed the performance advantages of spatial multitasking over cooperative multitasking using a simple even-split of GPU SMs among applications. In this section we 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 we were unable to simulate Oracle scheduling for more than two applications sharing the GPU.

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

Table 6. Summary of average interconnect latency for all combinations of applications.

	1 Apps	2 Apps	3 Apps	4 Apps
Geometric 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

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 no application is given more SMs than it can use based on the number of thread blocks in the program.

Rounds attempts to assign SMs such that there are fewer idle SMs near the completion of a kernel. In our simple implementation of spatial multitasking, 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. We 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) we can predict how many rounds it would take to execute a kernel. The Rounds heuristic starts with an even-split of SMs among applications. With the even-split we calculate the number of rounds each application would take to execute. After this, we find the minimum number of SMs an application can be assigned without increasing the number of rounds it would take to execute over an even-split of SMs. We then find which applications would benefit from adding SMs assuming other applications receive the minimum number of SMs just calculated. Finally, we chose the partitioning that results in the minimum total number of rounds summed over all of the applications. In the case of ties, we choose 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 2 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 per-

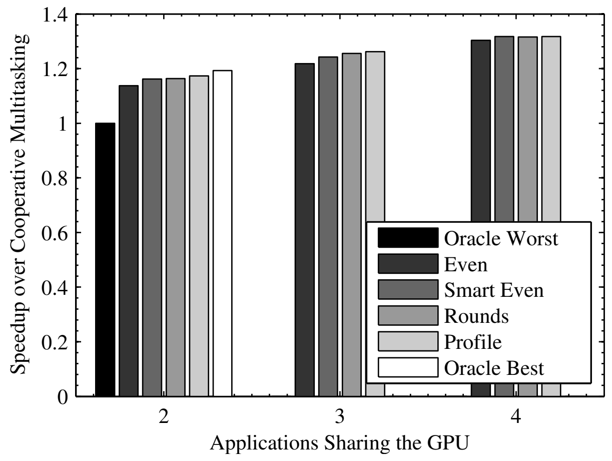

Figure 10. Average speedup of spatial multitasking over cooperative multitasking for several SM partitioning heuristics.

formance 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, the same input was used 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:

$$\sum_{i=1}^{N} S(i,j)^{\frac{1}{N}} \tag{2}$$

where $S(i,j)$ is the speedup of application i depicted in Figure 2 when j SMs are assigned to the application and N is the number of applications sharing the GPU.

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.

Figure 10 compares the performance of our 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, Rounds, and Profile is insignificant for this GPU configuration. When two applications share the GPU, these simple partitioning heuristics approach the optimal Oracle Best performance. The average performance of Oracle Worst is

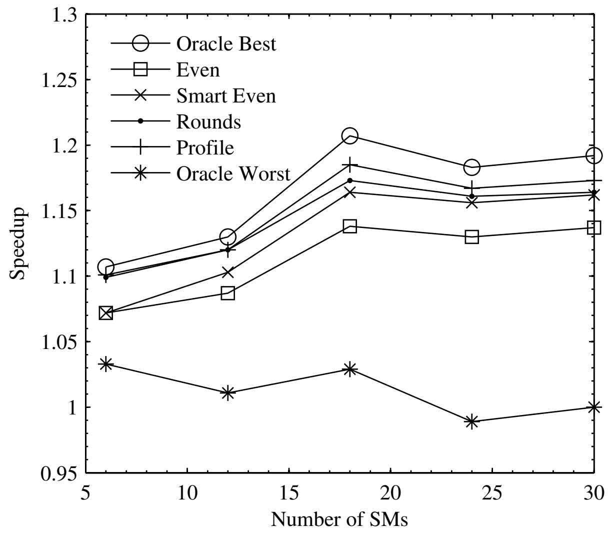

Figure 11. Average speedup of spatial multitasking compared to cooperative multitasking for several SM partitioning heuristics as the total number of SMs in the GPU is varied. Two applications share the GPU.

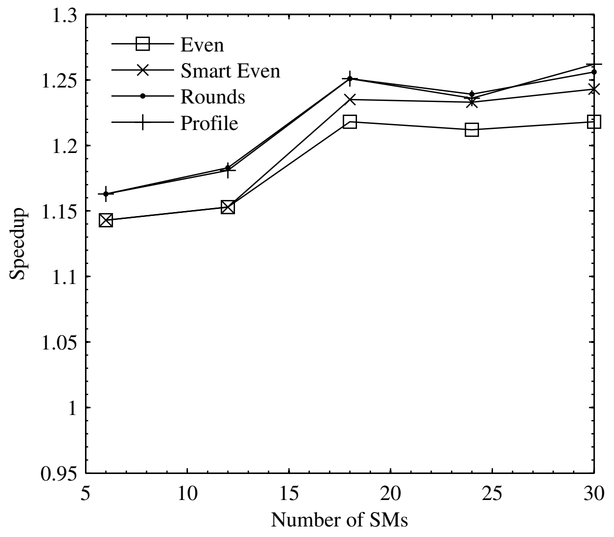

Figure 12. Average speedup of spatial multitasking compared to cooperative multitasking for several SM partitioning heuristics as the total number of SMs in the GPU is varied. Three applications share the GPU.

close to the performance of cooperative multitasking, which suggests that even with poor partitioning heuristics spatial multitasking can outperform cooperative multitasking.

Figures 11 and 12 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 Rounds and Profile. Profile requires 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 Rounds and Profile. In this case Rounds would be the best heuristic to choose because it performs just as well as Profile, but does not require an extra profiling step at install time.

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

4. Challenges of Spatial Multitasking

Currently, only a single application executes on the GPU at a time. Spatial multitasking changes this to allow GPU resources to be partitioned among multiple applications. However, there are several challenges for both hardware and system software that must be addressed to implement spatial multitasking. In this section we outline some of these challenges assuming spatial multitasking partitions SMs among applications. We plan to investigate more specific implementation details of spatial multitasking in future work.

The GPU memory system requires modifications to support multiple applications. Although several GPU architectures already virtualize global and texture memory for a single application [11], the memory sub-system will have to 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. In set-top and portable devices, memory requirements of applications are likely modest, making paging unnecessary; however, for desktop and scientific computing, memory requirements are likely higher. In situations where GPU resources become over-subscribed, such as running out of physical GPU memory when paging is not implemented, the system should fall back on a mixture of spatial and temporal multitasking.

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. For this reason, in spatial multitasking, the OS is best-suited for resource partitioning. The GPU would still be responsible for scheduling individual threads to SMs within the partitioning constraints the OS has set.

5. Related Work

Multitasking has been previously researched on several platforms that share properties with GPUs. Intel's Larrabee, a many-core architecture intended for graphics processing [25], would have fully supported preemptive and spatial multitasking for the general-purpose cores, like most other x86-based multicore CPUs. However, this work differs from Larrabee by investigating multitasking for standard GPUs, which have a much larger number of very simple cores than Larrabee. Additionally, the number of hardware threads supported on each GPU core is much higher than the four threads per core supported on Larrabee.

The Cell multiprocessor is made up of a general-purpose super-scalar processor and several SIMD processors [10]. 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 [9], but 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 this work targets, which presents different challenges for spatial multitasking.

NVIDIA's Fermi architecture supports concurrent execution of GPGPU kernels from a single application [19], but 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. This is partly caused by limitations imposed by data dependencies across kernels, a problem not present in spatial multitasking.

The issue of time-sharing versus space-sharing has been investigated on shared-memory multiprocessors as well [15, 21]. McCann *et al.* showed space-sharing scheduling policies out-perform time-sharing [15]. 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 differently from shared-memory multiprocessors, GPGPU applications still tend to have convex speedup versus resource curves, as we have shown in this paper. This feature is one of the motivating factors for GPU spatial multitasking. On the other hand, Ousterhout noted that applications exhibiting fine-grained inter-process communication perform poorly when only a subset of the communicating processes are executing concurrently [21]. To handle this situation, he proposed a technique that schedules processes from the same job to multiple processors simultaneously (i.e., co-scheduling), effectively time-sharing multiprocessors. In GPGPU architectures, fine-grained communication can only occur among small groups of threads, known as thread blocks in NVIDIA architectures. Thread blocks are executed in parallel on a single SM, eliminating the need for co-scheduling in GPGPU architectures.

6. Conclusion

We proposed 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. We 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. Our simulation results also indicated that spatial multitasking has the potential to offer significant performance benefits compared to cooperative multitasking by executing applications in parallel. Smart Even resource partitioning showed average speedups of 1.16, 1.24, and 1.32 over cooperative multitasking for two, three, and four applications sharing the GPU, respectively. By allowing applications to share the GPU, spatial multitasking out-performs cooperative multitasking by using available GPU resources more efficiently. Finally, spatial multitasking provides opportunities for new, flexible scheduling and QoS techniques that can further improve the user experience.

Acknowledgement

We would like to thank the anonymous reviewers for their valuable comments. This work is supported in part by an SRC grant (Task ID: 2080.001), an NSF CAREER grant (CCF-095360), and a generous gift grant from AMD.

References

[1] S. Al-Kiswany et al. StoreGPU: exploiting graphics processing units to accelerate distributed storage systems. In *HPDC '08*, pages 165–174, 2008.

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

[3] A. Bakhoda et al. Analyzing CUDA workloads using a detailed GPU simulator. In *ISPASS '09*, April 2009.

[4] S. Brodhead. GPU flame fractal renderer. http://sourceforge.net/projects/flam4/.

[5] CUDA zone – the resource for CUDA developers. http://www.nvidia.com/object/cuda_home.html.

[6] CUDPP: CUDA data parallel primitives library. http://gpgpu.org/developer/cudpp.

[7] GPGPU benchmarks. http://ercbench.ece.wisc.edu.

[8] GPGPU-Sim. http://www.gpgpu-sim.org.

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

[10] J. A. Kahle et al. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 2005.

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

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

[13] S. A. Manavski. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In *ICSPC '07*.

[14] Maxime. Ray tracing. http://www.nvidia.com/cuda.

[15] C. McCann, R. Vaswani, and J. Zahorjan. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 11(2):146–178, 1993.

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

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

[18] NVIDIA Quadro FX 5800. http://www.nvidia.com/object/product_quadro_fx_5800_us.html.

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

[20] A. Obukhov and A. Kharlamov. *Discrete Cosine Transform for 8x8 Blocks with CUDA*. NVIDIA Corporation, October 2008. http://developer.download.nvidia.com/compute/cuda/2_2/sdk/website/projects/dct8x8/doc/dct8x8.pdf.

[21] J. K. Ousterhout. Scheduling techniques for concurrent systems. In *ICDCS '82*, pages 22–30, 1982.

[22] J. D. Owens et al. GPU computing. In *Proceedings of the IEEE*, volume 96, pages 879–899, March 2008.

[23] Parboil benchmark suite. http://impact.crhc.illinois.edu/parboil.php.

[24] S. Pronovost, H. Moreton, and T. Kelley. WDDM v2 and beyond. WinHEC '06. http://download.microsoft.com/download/5/b/9/5b97017b-e28a-4bae-ba48-174cf47d23cd/PRI103_WH06.ppt.

[25] L. Seiler et al. Larrabee: a many-core x86 architecture for visual computing. In *SIGGRAPH '08*, pages 1–15, 2008.

[26] S. Sengupta et al. Scan primitives for GPU computing. In *GH '07*, pages 97–106, 2007.

[27] StoreGPU. http://www.ece.ubc.ca/~samera/projects/StoreGPU/code/.

[28] R. Szerwinski and T. Güneysu. Exploiting the power of GPUs for asymmetric cryptography. In *CHES '08*.

[29] W. J. van der Laan. GPU-accelerated Dirac video codec. http://diracvideo.org.

[30] S. Vaughn-Nichols. Vendors draw up a new graphics-hardware approach. *Computer*, 42(5):11–13, May 2009.

[31] 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*, pages 297–302, October 2009.