Task Allocation and Migration for Novel Architectures

By

Paula Aguilera Diez

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Electrical Engineering)

at the

University of Wisconsin-Madison

2016

Date of final examination: Monday, December 5th 2016

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

Katherine L. Morrow, Associate Professor, Electrical & Computer Engineering

Nam Sung Kim, Associate Professor, Electrical & Computer Engineering – University of Illinois

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

Karthikeyan Sankaralingam, Associate Professor, Computer Sciences

Xinyu Zhang, Assistant Professor, Electrical & Computer Engineering

AKNOWLEDGEMENTS

First, I would like to thank my advisors Katherine Morrow and Nam Sung Kim. They are the reason that I can graduate with this degree. Their patient and knowledge has always kept me motivated.

I would also like to thank the rest of my committee: Mikko H. Lipasti, Karthikeyan Sankaralingam and Xinyu Zhang. They took the time to learn about my work and their questions and comments helped me better understand the implications of my work.

David Roberts, Nuwan Jayasena and Dong Ping Zhang guided me during my internship AMD, where I had my first experience working outside the university. I really enjoyed this time and it encouraged me to keep pursuing computer architecture as a career.

Many of my fellow labmates have become good friends with the years; working and spending time with them was probably one of the best things during this time in school.

Last, a special thank you goes to my family. Although they were and still are very far, they have always been so close to me and they have always helped with anything that I needed. I cannot imagine what I would do without them.

ABSTRACT

In this thesis I study resource allocation and task migration for novel architectures. Nowadays in order to increase the performance of microprocessors, reducing their energy consumption has become essential as we hit the power wall. Novel architectures are being proposed that increase energy efficiency and thus provide higher performance per watt than traditional CPUs. Yet the differences between these architectures and traditional microprocessors require new approaches for assigning and migrating tasks across computational resources.

Among these new architectures are Graphic Processing Units (GPUs) and Processing-In-Memory (PIM). GPUs are highly parallel platforms that can offer both high performance and energy efficiency by using a large number of simple and energy-efficient compute elements. PIM architectures place the computation logic and the memory on the same chip reducing the energy of moving data from the memory to the processor. In the last few years, GPUs have been widely adopted for general-purpose computing on GPUs (GPGPU); these capabilities have been enabled thanks to their increased programmability and the emergence of high-level programing languages such as OpenCL [1] and CUDA [2].

With every generation, GPUs contain more computing resources; however, it has been shown that many GPGPU applications fail to fully utilize these resources [3]. Spatial multitasking has been proposed as a solution for this problem [3]. Spatial multitasking shares the GPU resources among multiple concurrently-executing applications. Traditionally, GPUs have been designed to run computations serially, one at a time, and thus they offer very poor spatial multitasking support. When multiple applications execute concurrently on the GPU, a decision needs to be made to determine how to distribute the computing resources among them. This

thesis presents and studies the effects of different resource allocations on performance when considering process variation, quality of service (QoS) and fairness.

On the other hand, PIM architectures try to solve a different problem: they reduce the large amount of energy that is spent moving data from main memory to the processor. Current PIM architectures place one or more DRAM dies atop (or under) a logic die implementing compute capabilities; this is done using 3D-die stacking. In the last part of this thesis I examine the feasibility of fine-grained work migration to reduce remote memory accesses in systems with multiple PIM devices. In such a system the memory is perceived as non-uniform; local memory accesses are more efficient than remote memory accesses. In this dissertation, I perform a high-level exploration to study when it is more efficient to perform a remote memory access versus work migration, and investigate mechanisms that have the potential to improve the efficiency of migration.

The mapping and re-mapping of computation to a subset of highly-parallel resources is critical to the utility of computing architectures, such as GPGPUs and PIM architectures, that are increasingly promoted as architectural solutions to increase performance while improving energy efficiency. The work presented in this thesis thus represents an important step in not only improving the capabilities of these architectures, but also their utility.

TABLE OF CONTENTS

FIGURES

TABLES

1 Introduction

Efforts to improve performance of current microprocessors are complicated by the power wall [4]. Power consumption is currently one of the most important constraints for microprocessor design, as high power consumption limits microprocessors' performance. New architectures that provide higher energy efficiency are being proposed, in order to continue to increase computing performance. General-Purpose GPUs (GPGPUs) and Processing-In-Memory (PIM) are two examples of novel architectures that try to mitigate the energy consumption problem.

GPUs provide high compute throughput and energy efficiency by using a large number of simple and energy-efficient compute elements. Although GPUs were initially developed for compute-intensive graphic applications, they have evolved considerably in the last few years. Increased programmability in the graphics pipeline [5] and the development of new high-level programing languages such as CUDA [2] and OpenCL [1] have made GPUs a popular platform to accelerate a wide variety of general-purpose applications, from scientific to multimedia applications [6] [7]. This area is known as GPGPU computing. Because of their high performance and relative energy efficiency, GPUs are being used in a large variety of computing systems from mobile to HPC.

As process technology keeps scaling, every GPU generation contains more computing resources. It has been shown that some GPGPU applications fail to fully utilize all the computing resources on a GPU [3]. One solution to this problem is spatial multitasking, where multiple concurrently executing applications share the GPU resources. This approach differs from the traditional method of multitasking in GPUs—cooperative multitasking—where the time to

execute is divided among the multitasked applications and the applications run in isolation on the GPU. Spatial multitasking provides higher resource utilization, responsiveness and overall system performance than cooperative multitasking, but requires effective methods to partition the GPU resources among concurrent applications. These decisions are complicated by problems such as process variation (which leads to different possible clock frequencies in different regions of the GPU), quality-of-service (QoS) requirements for any of the applications, and issues related to fair execution.

PIM architectures also attempt to improve energy consumption. The motivation for their design is based on the observation that data movement from the memory system to the processor consumes a large amount of the total energy in current computing systems [8], [9], [10]. Unfortunately, applications that present irregular memory access patterns and low locality cannot take advantage of the traditional cache hierarchy, making data access energy a significant concern [11], [12]. For these applications a large amount of the total system energy is spent in the memory system. A solution to this problem is to place the computation closer to its data; this way the energy of accessing the memory is reduced resulting in performance improvements.

A memory module with processing in Memory (PIM), or a *PIM stack*, implemented using 3D stacking technology places one or more memory dies atop (or under) a logic die implementing compute capabilities. I study a system that consists of a host processor and multiple such PIM stacks. Each in-memory processor has low-latency, high-bandwidth, and low-energy access to data in the *local* memory stacked on top of it. Accesses to *remote* data in other memory modules incur high latency and energy consumption and low memory bandwidth. A large number of remote memory accesses can negate the benefits of PIM. In these situations migrating work to execute on the PIM device co-located with the data that it accesses can be

more efficient than fetching the data from a remote PIM module, when the overhead of migrating work is lower than that of performing remote memory accesses.

GPGPUs and PIMs present new challenges when it comes to resource allocation and task scheduling and migration. Improving these capabilities for PIM and GPGPUs will not only result in performance and energy gains, but also better ways of providing QoS and fairness for different applications.

1.1 Contributions

This thesis presents several novel resource allocation techniques for GPGPU and PIM architectures, and studies their effects on performance and energy consumption.

I first consider a spatial multitasking GPU where multiple applications run concurrently on the GPU by using a subset of the computing resources. In this case, I examine how process variation, QoS requirements and fairness considerations can be used in the scheduling process to result in different tradeoffs. Later, I look at how to dynamically allocate tasks to the different GPUs on a system with multiple PIM stacks in order to exploit data locality and thus improve performance and energy consumption.

First, I look at process variation effects on GPGPU computation. As process technology scales below 65-nm, considerable transistor delay variations occur both within-die (WID) and die-to-die (D2D) [13]. Specifically, WID process variations result in a large difference among the maximum operating frequencies (F_{max}) of each core on the same die. Using a single global clock, as is the case in most CPUs and GPUs, requires all cores to run at the same frequency—the minimum of the cores' F_{max} values. Because GPUs have a much larger number of cores than CPUs, this issue is particularly problematic. Clocking each core at its own F_{max}, removes the

artificial restriction of global clocking (GC), and thus improves throughput. This technique is called per-SM clocking (PSMC) [14]. When multiple GPGPU applications with different characteristics run on a spatially-multitasking GPU with PSMC, the assignment of SMs to applications can significantly impact performance. Some applications benefit from higher SM frequencies, while others do not suffer when executing on SMs with lower frequencies. In this thesis, I exploit WID process variations and the characteristics of GPGPU applications to maximize the overall performance of spatial-multitasking GPUs. The main contributions of this project are presented below:

- I characterize the sensitivity of GPGPU applications' performance to SM's operating frequency in the context of spatial-multitasking GPUs.
- I present WID process variation-aware SM-to-application assignment techniques for spatial-multitasking GPUs supporting PSMC.
- I demonstrate that assigning faster SMs to compute-bound applications and slower SMs to memory-bound applications can considerably improve overall performance of spatial-multitasking GPUs supporting PSMC.
- I evaluate the efficacy of the proposed application assignment technique for various SM partitioning algorithms for spatial-multitasking GPUs.

Second, I study resource allocation when one or more of the concurrently-executing applications on the GPU have QoS requirements, as are common in many multimedia and other applications. Performing below an application's QoS requirement is considered undesirable, whereas performance above it may not offer any benefit. These applications may concurrently run on the GPU with other applications that do not have QoS requirements, but do benefit from increased performance. I refer to this latter class of applications as "best-effort" applications.

QoS requirements complicate GPU resource allocation for GPUs using spatial multitasking, but they also provide an opportunity. If QoS requirements can be satisfied without using all of the GPU's resources, the remaining resources can either be disabled to reduce power consumption, or allocated to any co-executing best-effort applications to increase their performance. The main contributions for this work are presented below:

- I show that GPU resources that are unused after QoS requirements are satisfied can be disabled to reduce power consumption or allocated to a co-executing best-effort application to increase its performance.
- I demonstrate that the resource requirement to satisfy an application's QoS depends on its characteristics and those of any co-executing applications. A runtime dynamic resource allocation algorithm is thus needed to maximize the benefit of a spatial-multitasking GPU running at least one QoS application.
- I propose a runtime dynamic resource allocation algorithm and evaluate its efficacy.

Third, I study the performance implications of considering fairness issues when allocating resources to concurrently-executing (spatially-multitasked) applications on the GPU. One straightforward allocation goal is to attempt to maximize overall system throughput and ignore fairness issues; however, if one application has significantly higher throughput per SM, that application will receive nearly all SMs, penalizing other application(s) executing on the GPU. Lower-throughput applications can suffer a significant penalty relative to the performance that they would have experienced when temporally-multitasked on the GPU using cooperative multitasking. A possibly more fair allocation of resources may be to evenly distribute the resources among the simultaneously-running applications, but this approach may not make the most effective use of resources, penalizing overall throughput. In this thesis I present different

allocation goals that aim to provide a balance of fairness and performance. The main contributions of this project are presented below:

- I present a set of different fairness policies to allocate resources to multiple concurrently-executing applications on a spatial multitasking GPU.
- I study the effects of these policies on the performance of individual applications and the system as a whole.
- I demonstrate a run-time approach for choosing a resource partition based on the chosen fairness policy.

Last, I explore the potential benefits of migrating work to bring computation closer to its data in Processing-in-Memory (PIM) systems. The systems I examine consist of multiple PIM stacks, where each in-memory processor only has efficient access to the memory on its same stack. Direct accesses to data in other memory modules — remote data accesses — can be highly inefficient. Therefore, it is sometimes beneficial to migrate work to the PIM stack where the remote data is located, rather than fetching remote data. This thesis studies techniques to determine when work migration is beneficial, and mechanism to make migrations more efficient. The main contributions of this project are presented below:

- I present a queueing framework to perform light-weight fine-grained task migration amongst PIM devices.
- I introduce hardware mechanisms that implement efficient queuing for PIM systems.
- I present a parameterized high-level timing model to estimate the performance of graph algorithms implemented atop this framework.

- I use the above timing model to study the possible performance tradeoffs of work migration under a variety of system configurations as well as application and graph characteristics.

1.2 Thesis Organization

The rest of the document is organized as follows: In Chapter 0 I introduce background and related work regarding GPGPU architecture and spatial multitasking, process variation, QoS, fairness, graph processing and task migration, and processing in memory. In Chapter 3 I describe my experimental framework for the GPGPU studies presented in this dissertation. In Chapter 4, I introduce my work to take advantage of process variation when allocating GPGPU resources to multiple concurrently-running applications on a GPU that supports spatial multitasking. In Chapter 5, I demonstrate how resource allocation can take advantage of GPGPU spatial multitasking to meet applications' QoS requirements while either reducing energy or increasing the performance of best-effort applications when possible. In Chapter 6, I discuss fairness and its implications in spatially-multitasked GPU resource allocation. In Chapter 6 presents my work in the area of task migration for graph algorithms using PIM in order to maximize locality and improve performance and energy consumption.

2 Background and Related Work

2.1 GPGPU Architecture and Execution

A GPU's specific architecture varies by manufacturer, but the main components of a GPU most commonly are: multiple cores (called "streaming multiprocessors" (SMs) in NVIDIA's GPUs), on-chip memory/interconnect, and off-chip DRAM. Because the experimentation for this dissertation focuses on NVIDIA architectures I use their terminology throughout the document; however, the same ideas could be applied to other GPGPU architectures supporting spatial multitasking.

Due to the large number of cores, GPUs are highly parallel and thus powerful computing platforms for not only graphics processing but also general-purpose computations [6]. Applications usually consist of a combination of GPU and CPU execution, with the most compute-intensive portions accelerated by the GPU; these parallel and compute-intensive sections of an application are implemented to run on the GPU and are called *kernels*. A General Purpose GPU (GPGPU) application can contain one or more of these kernels. The CPU part of the application launches the kernels when needed, and may execute outer control-flow operations if they cannot be implemented efficiently in the GPU.

A GPGPU kernel consists of a large number of threads organized in a hierarchy of independent thread blocks that form a grid. A thread block is a group of concurrently-executed threads that can communicate through the shared memory and can synchronize. A thread block executes completely in a single SM and each SM can execute one or more thread blocks concurrently. A kernel launch is followed by initialization of its threads, after which the global thread block scheduler on the GPU starts distributing thread blocks to the available SMs; once no

SM has available resources for more thread blocks, the scheduler will wait until one of the currently-executed thread blocks finishes execution.

2.2 GPGPU Multitasking

Current GPUs only have basic multitasking capabilities when executing applications. Generally, GPU applications must wait to execute until the application currently occupying the GPU voluntarily yields control—a form of cooperative multitasking. In contrast, an operating system (OS) typically preemptively multitasks a CPU, 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.

However, GPUs are highly-parallel compute devices with an ever growing number of cores, and past research showed that some GPGPU applications fail to fully utilize GPU resources [3]. This trend is likely to continue with increasing GPU compute capabilities from advances in GPU design. This motivates the use of *spatial* multitasking (dividing resources amongst co-executed applications) in addition to *temporal* multitasking (dividing resource time amongst applications so they appear to execute simultaneously). Thus, multiple applications that might each under-utilize the GPU can execute simultaneously, improving GPU utilization and thus overall system throughput [3]. Figure 1 is an example of three applications running on the GPU using spatial

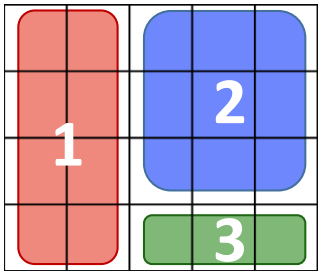


Figure 1: Spatial multitasking on the GPU with three applications sharing the computing resources. Each application gets a share of the resources.

multitasking, each on a subset of the GPU resources.

Multi-core CPUs use spatial multitasking when they execute multiple applications simultaneously on different cores. However, this form of multitasking is currently very limited on GPU architectures. For example, NVIDIA's Fermi architecture supports spatially-multitasking kernels, but only if they belong to the same application [15]. NVIDIA's newer architectures, Kepler [16] and Maxwell [17] support co-execution of multiple GPGPU kernels from different applications, as long as there are resources available on the GPU, but the implementation details have not been disclosed. Furthermore, it is not yet clear how the resources are allocated to the different kernels, and how performance is impacted by these allocation decisions.

As the GPU moves onto the same chip as the CPU [18], [19], the potential advantage of using GPUs as parallel accelerators with multitasking capabilities will grow. Fully exploiting GPUs will require new temporal and spatial GPU multitasking techniques. Although spatial multitasking provides more effective use of GPU resources, leading to improved throughput and system responsiveness, many issues require further research.

2.3 Process Variation

As process technology scales below 65-nm, considerable transistor delay variations occur both die-to-die (D2D) and within-die (WID) [13]. D2D variations affect all transistors on a die identically, whereas WID variations result in different characteristics for transistors across a single die. When there are multiple cores in a die, WID process variations result in cores having a different maximum operating frequency (F_{max}). As technology scaling encourages integrating more cores per die, WID variations lead to more significant core-to-core (C2C) frequency

variations. Using a single global clock, as is the case in most CPUs and GPUs, requires all cores to run at the same frequency—the minimum of the cores' F_{max} values. Because GPUs have a much larger number of cores than CPUs, this issue is particularly problematic. Clocking each SM at its own F_{max}, removes the artificial restriction of global clocking (GC), and thus improves throughput. This technique is called per-SM clocking (PSMC) [14]. There are some applications that benefit from running on SMs with higher frequencies, while others do not suffer when executing on SMs with lower frequencies. When multiple GPGPU applications with different characteristics run on a spatially-multitasking GPU with PSMC, the assignment of SMs to applications can significantly impact performance.

Technology scaling increases WID, and also encourages integrating more SMs per die, resulting in larger SM-to-SM frequency (and leakage power) variations. According to a model applied to a GPU comprised of 30 SMs in 32nm technology, the fastest SM can be up to 40% faster than the slowest SM [20]. In a globally-clocked GPU this leads to the "worst of both worlds": all SMs are limited to the F_{max} of the slowest SM, but cores otherwise capable of higher frequencies consume much more leakage power, degrading power efficiency considerably [13]. In contrast, PSMC allows us to exploit SM-to-SM F_{max} variation to improve overall performance [14].

This thesis extends prior work [3] by exploiting WID process variation, PSMC, and an SM partitioning methodology that assigns kernels to SMs based on workload characteristics and SM frequency. I exploit this WID F_{max} variation to improve performance by implementing PSMC on a GPU. Per-core PLLs are already commonly used even for globally-clocked multiprocessors to optimize clock skew and reduce power dissipation of clock trees [21]. Thus, I assume there would be little to no extra cost to support PSMC using per-SM PLLs on a GPU. If the number of

SMs is such that it makes the cost of having a PLL per SM too high, a PLL per cluster of neighboring SMs could be used to lower the overhead. This approach would still provide benefit, since WID process variations are typically dominated by spatially correlated ones.

K. Bowman *et al.* [22] and Humenay *et al.* [23] examined the impact of WID C2C frequency variations on the performance of multi-core processors. The performance improvement of a *frequency islands* (FIs) scheme exploiting WID C2C frequency variation across a range of multicore processor designs is examined using an analytical throughput model of the FIs multi-core processors [20]. Lee *et al.* [14] showed that PSMC on a GPU having WID process variation can significantly improve performance because there is rare or no inter-SM communications in most GPGPU kernels. The independence of the thread blocks within the same GPGPU kernel allows any available SM in a GPU to execute multiple blocks independently while running at a different frequency.

2.4 Quality of Service (QoS)

Many multimedia and other applications have certain quality-of-service (QoS) requirements. Performing below the application's QoS requirement is considered undesirable, whereas performance above it may not offer any extra benefit. For example, encryption algorithms might need to keep pace with communication algorithms or media playback, and algorithms such as JPEG might be required to decode frames at a certain rate at which they are presented to the consumer. Applications with QoS requirements may concurrently run on the GPU with other applications that do not have QoS requirements, but do benefit from increased performance. I refer to this latter class of applications as "best-effort" applications.

Allocating limited resources to multiple applications satisfying their QoS requirements has been studied before; low level parameters descriptive of the task such as arrival rate, period or service time are usually considered (*e.g.*, [24], [25]). For this work I take into consideration a single QoS dimension, which is the IPC executed by the QoS application.

Harada *et al*. looked at adjusting the QoS requirements of an application to reduce its energy consumption; this work targeted embedded real-time systems (*e.g.*, [26]). In this thesis I also study using QoS requirement information to allow energy consumption to be reduced, but without changing the QoS requirements. Other previous research studied how to satisfy the required level of QoS for a certain application while guaranteeing the lowest energy consumption by adjusting the supply voltage and frequency on a CPU with DVFS support (*e.g.*, [27]). I pursue the same objective, considering a GPU, but instead by minimizing the amount of resources assigned to a QoS application running on the GPU, while still satisfying its performance requirements.

Hong and Kim use an analytical GPGPU model to find the number of cores in the GPU that provides the largest performance per watt for a certain application [28]. Although they consider leaving resources (GPU cores) idle to save power, they do not take into account QoS; in this work I also consider how to partition the GPU resources between multiple QoS and best-effort applications running on the same GPU to minimize power consumption or maximize performance of a co-scheduled best-effort application.

TimeGraph schedules GPU commands from multiple applications that try to access the GPU; it guarantees the performance of certain applications by giving preference to its commands

[29]. My approach instead satisfies QoS by partitioning the GPU resources among applications running concurrently in a spatial multitasking GPU.

2.5 Fairness

Previous work exists that studies GPU resource management for multitasking systems, but it usually focuses on temporal multitasking (dividing the time among the applications to execute on the complete GPU resources). *GERM* is another GPU scheduler that guarantees fairness in the system by assigning an equal amount of time to the competing applications to run on the GPU [30]. In contrast, the approach that I present in Section 0 (which is based on spatial multitasking) partitions the GPU computing resources among competing applications and executes them simultaneously, each on their assigned subset of GPU resources.

Jog *et al*. [31] look at the issue of fairness when multiple GPGPU applications run concurrently on the GPU (spatial multitasking). They show that GPGPU applications with high memory intensity can negatively impact the performance of other GPGPU applications. To solve this problem they modify the memory system to serve memory requests from the different applications in a round robin fashion; now applications with lower memory intensity are not starved by other applications that present higher memory usage. While their proposal reduces the negative interactions that can occur from multiple applications sharing the memory system, it does not completely solve the fairness problem, nor does it handle fair allocation of compute resources as I propose in this work.

Other previous work focuses on fairly distributing computing resources among concurrent executing applications in simultaneous multi-threaded (SMT) and chip multiprocessor (CMP) systems. Bhadauria *et al*. study when to co-schedule threads from multiple parallel programs to

the individual processors in a CMP. They try to schedule threads that present a different use of resources to run concurrently, optimizing for both overall system performance and individual thread energy-delay [32]. Although their work focuses on the advantages of space-scheduling vs. time-scheduling, they also consider fairness. They limit their co-scheduling to the applications whose performance does not scale well with the number of threads simultaneously executing from that application; those applications that do scale well are run in isolation. This means that the per-thread performance of a co-scheduled application actually increases, despite the fact that fewer of its threads can execute simultaneously. However, the inability to co-schedule applications from both categories (those that do and those that do not scale well with increasing thread execution) limits the possible solution space.

Another concern regarding fairness in multiprocessor (SMT and CMP) systems is that contention in the shared resources (such as memory) can lead to reduced performance relative to running in isolation; hardware solutions that assign a share of the resource to each of the applications competing for it, provide some amount of fairness. Nesbit *et al.* [33] present a memory scheduler that assigns each application in the system the same amount of memory bandwidth, independently of the resource usage of the other applications; Kim *et al*. use hardware mechanisms to partition the L2 cache, this way all threads running on the system are affected equally by the sharing of the L2 [34]. Cazorla *et al*. study how to reduce the performance variability that an application suffers when running on an SMT CPU with a variety of co-scheduled applications; in this case resources such as the issue queues, registers and fetch priority are divided among the concurrent applications [35]. Ebrahimi *et al*. [36] study how individual application's performance is affected when multiple applications run on a CPM sharing the memory system. They propose clocking down the cores that place more pressure on

the shared resource and negatively impact other application's performance. The clocked down cores issue fewer memory requests reducing the pressure on the memory system, and applications that were negatively impacted by the sharing experience performance improvement. This particular method could be used in conjunction with my proposed methods to control fairness using resource allocation.

2.6 Processing in Memory and 3D Die Stacking

The performance gap between the processor and the memory system has kept increasing over the years [9]. Currently the memory system presents higher latency and energy consumption than the processor [37]; which results in the memory system being responsible for a large portion of the total system energy consumption [8], [10]. This trend is particularly problematic for applications that are memory bound and present irregular memory access patterns and low data reutilization; for these applications accessing data is expensive and inefficient. A solution to this problem is to place the compute logic and the memory closer together, reducing the energy consumption and latency of accessing data from the memory to the processor [38], [39], [40].

Processing in memory (PIM) architectures couple compute logic and memory on the same chip. Traditional PIM proposals integrated logic and memory on the same chip using the same process technology; this results in reduced performance and it is expensive to produce due to the low yield. This is the reason why PIM has not been adopted until now.

This thesis studies PIM architectures implemented with 3D die stacking. 3D stacking is a technology that overcomes the previous problem of integrating both logic and memory in the same chip using the same process technology; 3D stacking places two or more dies on top of each other, and makes it possible for the different dies to use different process technology.

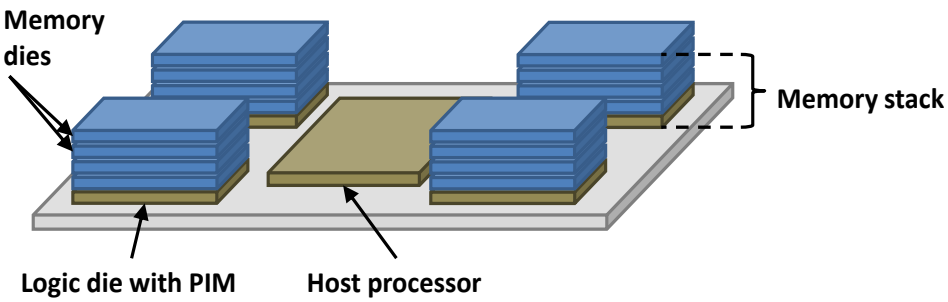


Figure 2: Example system with a host processor and multiple memory devices with PIM capabilities. Figure taken from"Thermal Feasibility of Die-Stacked Processing in Memory", by Yasuko Eckert, Nuwan Jayasena, Gabriel H. Loh, in In the 2nd Workshop on Near-Data Processing (WoNDP), 2014.

Previous research has shown that the main benefit provided by 3D stacking is the high memory bandwidth access that the processor has to the memory [41]. There are industry standards that show the interest in 3D die stacking; the hybrid memory cube (HMC) and High Bandwidth Memory (HBM) are examples of it [42], [43]. As the main benefit of 3D stacking is the large memory bandwidth, applications with high bandwidth demand will be a good fit for PIM systems [41].

The system that I study is shown in Figure 2; there is a host processor connected to multiple memory devices with PIM capabilities. In such a system, memory intensive computations are offloaded to the PIMs and computation and data are partitioned among the stacks. Due to the low latency and high bandwidth to the memory available to the processor when accessing data in its local stack and the high latency and high energy consumption of remote memory accesses, maximizing data locality is desired. However, emerging workloads with increasingly irregular memory access patterns make it difficult to place the data and computation in the same stack, resulting in computation having to access data that is remote. Migrating work to execute on the stack where the remote data is located, can be more efficient that fetching the data from a remote stack.

Previous research exists on the topic of work and task migration for distributed systems (*e.g.*, [44], [45], [46], [47]). However, PIM introduces a number of new considerations that warrant revisiting the topic. The PIM stacks within a single system, in some ways, resemble non-uniform memory access (NUMA) architectures. However, the close coupling of PIM to main memory makes local memory access extremely inexpensive from both performance and power perspectives relative to traditional NUMA machines and other forms of multi-processor systems. Further, the PIM stacks can be interconnected via memory interfaces enabling efficient, fine-grain communication (*e.g.*, a single cache line) among them. In addition, PIM's proximity to memory enables highly efficient, hardware-assisted implementations of queue primitives. These factors justify and enable a finer granularity of tasks than many of the prior studies. In this study, I consider work migration at the granularity of the processing performed on a single vertex of graph algorithms.

2.7 Large-Scale Graph Processing

A graph is a data structure that tracks the connections between the different elements in a set. Many important applications these days such as social science and machine learning use graphs in their computations; some of the graphs that these applications work with are very large having at least millions of vertices. Graph algorithms such as breadth-first search, connected components, and shortest path are used frequently in other more complex applications. Graph algorithms are memory-bound and present irregular and graph-topology-dependent memory access patterns with limited data locality; these characteristics result in graph algorithms having high memory bandwidth requirements [48]. These characteristics also make graph algorithms a good match for PIM architectures. However, the irregular memory access patterns make it challenging to partition the computation and data across the multiple PIM stacks so that

computation is co-located with its data and remote memory accesses are avoided. Therefore, I explore work migration techniques targeted towards graph processing algorithms to improve their performance in systems with multiple PIM stacks.

Narse *et al.* have compared two different approaches used to parallelize graph algorithms: topology- and data-driven [49]. In the first one all the vertices are visited regardless of whether there is work to perform on the vertices. In the second approach vertices are only visited if they are active and there is work to be done on them. This second implementation requires using a worklist data structure to track the vertices that need processing; threads read active vertices from the worklist and if, during the processing of a vertex, more vertices become active, they are written to the worklist. Updates to the worklist need to be atomic to prevent conflicts among concurrent updates. While data-driven implementations are more work-efficient, they can suffer from contention when multiple threads are updating the shared worklist [49]. In this thesis I propose hardware mechanisms that take advantage of PIM's proximity to memory to implement efficient queues and reduce the overhead of using work migration.

In cases where a large graph is distributed across multiple memory modules, I anticipate the common case is for all PIM devices to execute the same code on different subsets of data. Therefore, task migration in these cases does not require moving code between devices as the code base is identical and is already available at each PIM stack. Task migration simply involves communicating additional vertices, edges, or other data elements to be processed at a remote stack. Therefore, migrating a task can be as simple and as low overhead as communicating a single word (*e.g.*, a vertex ID) to a remote PIM device. In many ways, this is similar to the original notions of *active messages* (*i.e.*, the data from the message is integrated into the ongoing

computation at the remote location) [50] and does not require the migration of a full register set or context state.

Many distributed systems that perform computations on graphs follow this same approach: all the vertices to a remote processor are collected and then sent in a package to the corresponding remote processor. This approach results in an additional communication time that can reduce system performance [44], [51], [52], [53]. My proposed fine-grained work migration overlaps the communication with the computation, improving performance.

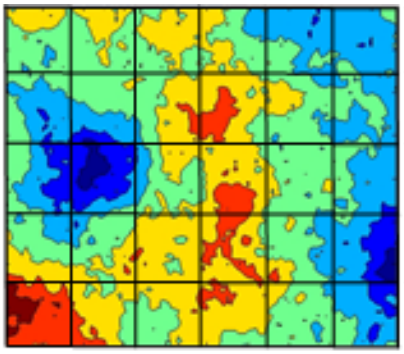
Hardware approaches have been proposed to improve the performance of data-driven graph algorithm implementations by reducing the contention on the shared data structures [54], but these often require dedicated hardware to hold the queue's data or metadata or both. As a result, these schemes incur high overheads to provision these dedicated structures and scalability is still limited by the hardware availability. Dedicated hardware also complicates context switching as the dedicated hardware must either be included in the context state or continue to be occupied by inactive contexts. Dedicated accelerators have also been proposed in the context of PIM [55], [48]. I use general purpose processors, which are able to execute a variety of applications not only graph algorithms.

Lock-free structures have also been proposed based on the computation of prefix-sums of multiple, parallel agents accessing the queue (so that each knows which location within the queue to access in parallel) [56]. However, these are difficult to orchestrate over loosely-coupled execution engines, as is the case with multiple processors that may not reside on the same chip. My proposal does not require extra time to compute the prefix-sums, and it also does not require coordination among the different PIMs in order to coordinate the queue operations.

3 Experimental Framework for GPGPU Studies

This section describes the experimental framework used in Chapters 0, 5, and 0 for my research related to GPGPUs. For all the experiments I use GPGPU-Sim, which is a detailed cycle-level execution-driven GPGPU simulator, I modify GPGPU-Sim to support spatial multitasking. For early experiments (resource allocation for spatial multitasking GPGPUs taking in consideration process variation and QoS) I used a modified version of GPGPU-Sim 2.1.1 [57] that models the NVIDIA Quadro FX 5800 GPU (GT200 architecture) with 30 SMs and 8 memory controllers [58]. For the later experiments (resource allocation for spatial multitasking GPGPUs considering fairness) I use GPGPU-Sim 3.2.1, which approximates the NVIDIA GeForce GTX 480 GPU based on the newer NVIDIA Fermi architecture with 16 SMs and 6 memory controllers [15]. Although I model NVIDIA GPUs, these results can be generalized to other architectures. Table 1 lists the parameters used with GPGPU-Sim to model these architectures.

I use the methodology presented in [59] to compare performance of multitasking GPGPU applications. I run all spatial multitasking simulations for 5M GPU cycles and measure the number of instructions simulated for each application. If an application completes in less than 5M cycles, it is restarted so that I am able to gather data for a full 5M cycles. I measure the number of instructions that each application completes, which represents a measure of work accomplished by the application during the 5M cycle timeframe. I then run the applications for the same number of instructions in cooperative multitasking, and determine the change in time required for those instructions. This ensures that I compare the same amount of steady-state work for each application between spatial and cooperative multitasking.



1.17	1.25	1.19	1.19	1.28	1.34
1.30	1.25	1.12	1.13	1.23	1.35
1.29	1.39	1.18	1.07	1.21	1.28
1.11	1.16	1.15	1.11	1.16	1.40
1.00	1.09	1.15	1.13	1.17	1.28

(a) (b)

Figure 3: (a) A WID Vth variation map for a 30-SM GPU and (b) the corresponding normalized F_{max}

Figure 3(a) shows a WID threshold voltage (V_{th}) variation map for a region (containing 30 SMs) of a die in a GPU. To generate spatially-correlated V_{th} and effective channel length (L_{eff}) maps for a GPU die, I used an analytical WID variation model and the following parameters as presented in [20]: WID correlation distance coefficient (0.5) and WID V_{th} variation (6.4%). I partitioned each variation map into 96×96 grid points and obtained a pair of V_{th} and L_{eff} values for each grid point using the analytical WID variation model. Then, I applied the corresponding pair of V_{th} and L_{eff} values to the 32nm predictive technology model [60] to obtain F_{max} of each grid point, which is modeled with a 24-stage FO4 inverter chain. Measuring propagation delay of this multi-stage FO4 inverter chain through HSpice simulation is a widely-accepted methodology to evaluate digital circuit performance [61]. Note that an SM's F_{max} is determined by the slowest grid point in the SM [20].

In Figure 3(b), each rectangle represents an SM and each number in a rectangle corresponds to F_{max} of each SM at the supply voltage (V_{DD}), normalized to the F_{max} of the slowest SM. Note that the F_{max} of the fastest SM is 1.4 times the F_{max} of the slowest SM in the GPU die shown in Figure 3(b). As more SMs are integrated in a die (*i.e.*, the die size becomes larger), I observe that the relative SM-to-SM F_{max} variations across the die increase.

Table 1. Key Hardware Parameters

Parameters	Quadro FX5800	GTX 480
# SMs	30	16
SM Frequency	1300 MHz	1400 MHz
Warp Size	32 Threads	32
SIMD Width	8	16 x 2
# Threads per SM	1024 (32 warps)	1536 (48 warps)
Registers / Shared Memory per SM	64 KB / 16 KB	32768 / 48 KB
# Memory Controllers	8	6
Memory Frequency / Bandwidth	800 MHz / 102 GB/s	924 M Hz / 134 GB/s
Bandwidth per Memory Module	8 Bytes per Cycle	
Interconnect Topology / Frequency	Crossbar / 650 MHz	Crossbar / 1400 M Hz
L1 cache (per SM)	None	16 KB
L2 cache (per SM)	None	768 KB

Table 2. Key Application Characteristics

Application	Category	Kernels	Thread blocks / kernel	Threads / thread block	Thread blocks / SM
AES Decryption [70] (AES)	Encryption	1	4097	256	6
AES Encryption [70] (AESE)		1	1420	256	2
SHA1 [75, 71] (SHA1)		1	1539	64	8
JPEG Decoding [72] (JPEG)	Image processing	1	512	64	8
Image Denoising [73] (ID)		1	4096	64	8
Ray Tracing [74] (RAY)	Graphics	1	2048	128	5
Fractals		2	512, 336	64, 256	8, 4
Streamcluster [76] (SC)	Data mining	1	128	512	3
Pathfinder [76] (PATH)	Grid traversal	1	463	256	6
Dirac Video Codec [82] (DVC)	Video processing	14	230 (avg.)	231(avg.)	4 (avg.)
Sum of Absolute Difference [81]		3	1584, 99, 99	61, 128, 32	8, 8, 8
Neural Network [77] (NN)	Artificial Intelligence	4	168, 1400,	169, 25, 1, 1	5, 8, 8, 8
Back Propagation [62] (BPR)		2	4096, 4096	256, 256	4, 4

To evaluate the performance of this research I use a variety of CUDA kernels and applications taken from ERCBench [3] and Rodinia [62] benchmark suites. Table 2 shows the most important characteristics of these applications.

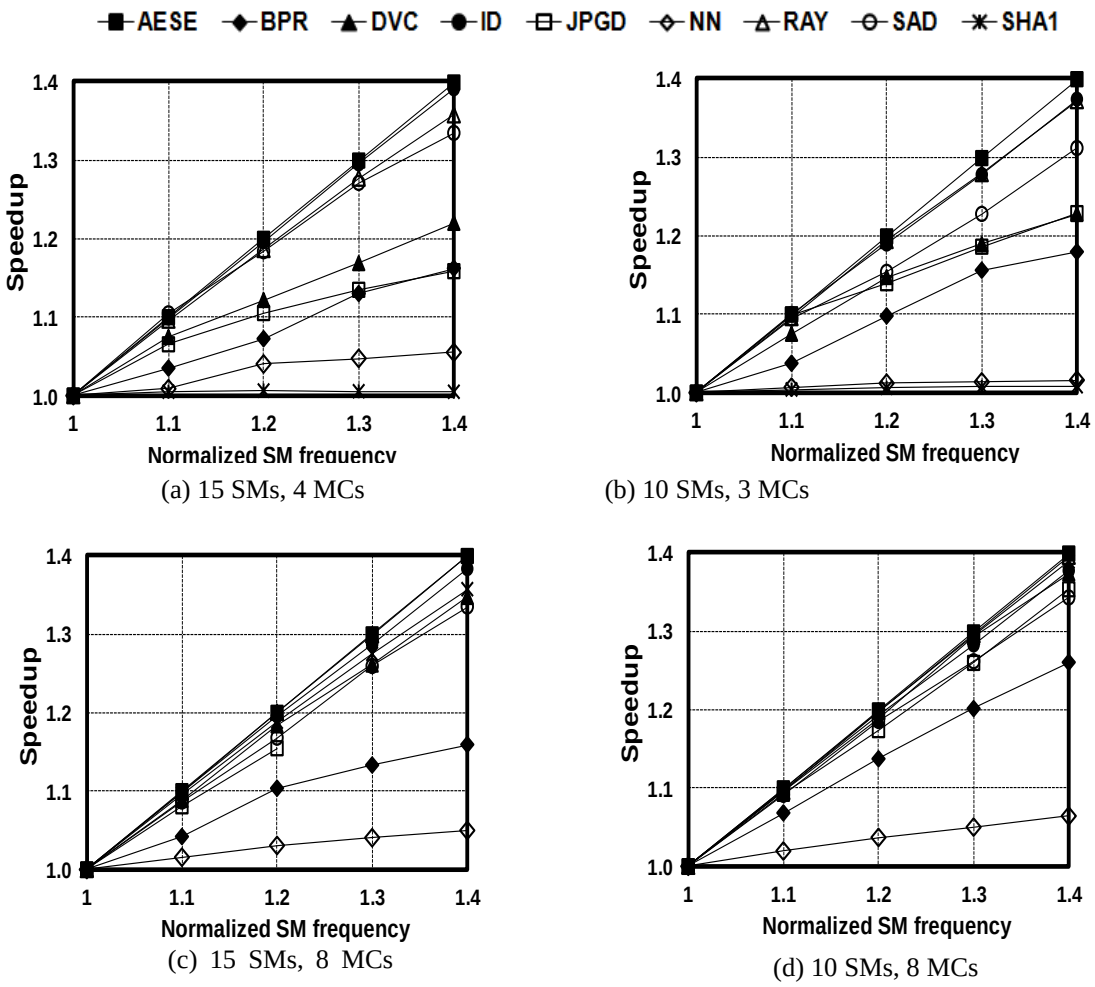
4 Process Variation-Aware Workload Partitioning Algorithms for GPUs Supporting Spatial Multitasking

Within-die (WID) process variation results in different F_{max} across the streaming multiprocessors (SMs) in a GPU. Per SM clocking (PSMC) provides the opportunity to assign SMs with different F_{max} to concurrently-executing applications. In this Chapter I study the benefits of using PSMC on a spatial multitasking GPU. For this work I model a NVIDIA Quadro FX 5800 GPU, which has 30 SMs and 8 memory controllers (MCs).

4.1 Application Characterization

I first study how sensitive different GPGPU applications are to frequency; I run each application several times in isolation on the simulator increasing the frequency of all the SMs each time. I repeat this for four different simulated architectures related to the NVIDIA Quadro FX 5800 GPU that allow us to estimate spatial multitasking performance from isolation data. I examine one architecture with 15 SMs and 4 memory controllers, one with 10 SMs and 3 memory controllers, and finally both 15 and 10 SMs with 8 memory controllers. The latter two architectures help identify if each application is more affected by SM frequency, memory bandwidth, or a combination of the two. I vary the frequency from 1.3GHz to 1.82GHz in 130MHz steps. All SMs are clocked at the same frequency in each run (*i.e.*, GC) and the highest frequency is 1.4 times the lowest frequency. This is a realistic frequency range considering WID process variations in a large die for 32nm technology, as previously discussed.

Figure 4 shows, for each application in isolation, speedups for different clock frequencies normalized to 1.3GHz. Based on how application's performance scales with SM frequency, I classify the applications in two groups: *compute*- and *memory-bound*.



(a) 15 SMs, 4 MCs

(b) 10 SMs, 3 MCs

(c) 15 SMs, 8 MCs

(d) 10 SMs, 8 MCs

Figure 4: Speedup vs. SM frequency for 15 and 10 SMs and 8, 4 and 3, memory controllers. Results are normalized to a GPU with SMs operating at the base frequency (1.3GHz). The X-axes is the normalized SM frequency.

AES-E, ID, RAY and SAD present almost linear speedup in all cases, showing compute-bound behavior. Applications such as DVC and JPG-D show moderate performance improvement with frequency, though the improvement is higher when more memory bandwidth is available. With 10 SMs and three MCs, DVC performs 23% better with 40% higher frequency. By increasing the number of MCs to eight, the performance of DVC increases by 37% (close to linear). SHA1's performance is also dependent on the number of MCs; at 1.82GHz with 10 SMs and three MC, it shows no speedup, with eight MCs the performance is 39% higher. NN presents little speedup with frequency, no matter the memory bandwidth available.

Table 3. Speedup of two-application workload using CSMF and CFMS assignments (Profile Partitioning, normalized to global clocking cooperative)

Workload	CSMF	CFMS	Δ	Workload	CSMF	CFMS	Δ
AES-E / JPG-D	1.34	1.38	0.04	ID / DVC	1.44	1.48	0.04
AES-E / NN	1.82	1.98	0.16	ID / SHA1	1.30	1.38	0.08
AES-E / BPR	1.57	1.60	0.03	RAY / JPG-D	1.29	1.33	0.04
AES-E / DVC	1.39	1.40	0.01	RAY / NN	1.56	1.67	0.12
AES-E / SHA1	1.28	1.35	0.07	RAY / BPR	1.48	1.52	0.04
SAD / JPG-D	1.32	1.34	0.02	RAY / DVC	1.40	1.42	0.02
SAD / NN	1.32	1.45	0.13	RAY / SHA1	1.19	1.24	0.05
SAD / BPR	1.30	1.35	0.05	BPR / NN	1.12	1.13	0.01
SAD / DVC	1.31	1.33	0.02	AES-E /ID	1.24	1.24	0.00
SAD / SHA1	1.08	1.13	0.05	JPG-D / DVC	1.36	1.36	0.00
ID / JPG-D	1.36	1.42	0.06	RAY / SAD	1.27	1.27	0.00
ID / NN	1.70	1.86	0.16	DVC / SHA1	1.17	1.18	0.01
ID / BPR	1.51	1.58	0.07	Geo Mean	1.36	1.41	0.05

I classify RAY, AES-E, ID, and SAD as compute-bound because they show nearly linear speedup with frequency. On the other hand, I classify SHA1, JPG-D, NN, BPR, and DVC as memory-bound since they do not benefit significantly from increased SM frequency. This classification can be done either in advance or at runtime by using a sampling method [63].

4.2 Resource Partitioning Algorithms

The following sections present experiments that combine PSMC with my SM partitioning algorithms to improve overall system performance in a spatial-multitasking GPU.

I use the following partitioning algorithms [3] to assign SMs to GPGPU applications and evaluate PSMC: *Even-Split* assigns SMs evenly among running applications, regardless of their computational needs. *Profile* uses isolated application profiling information to choose the best SM partitioning for each application. In my experiments, I profile applications in advance, and the results are used by the allocator. However, the profiling could be performed at runtime [63] if data-dependent application behavior is a concern. I use *Even-Split* and *Profile* to examine SM partitioning methods that have different objectives: *Even-Split* aims to provide applications with

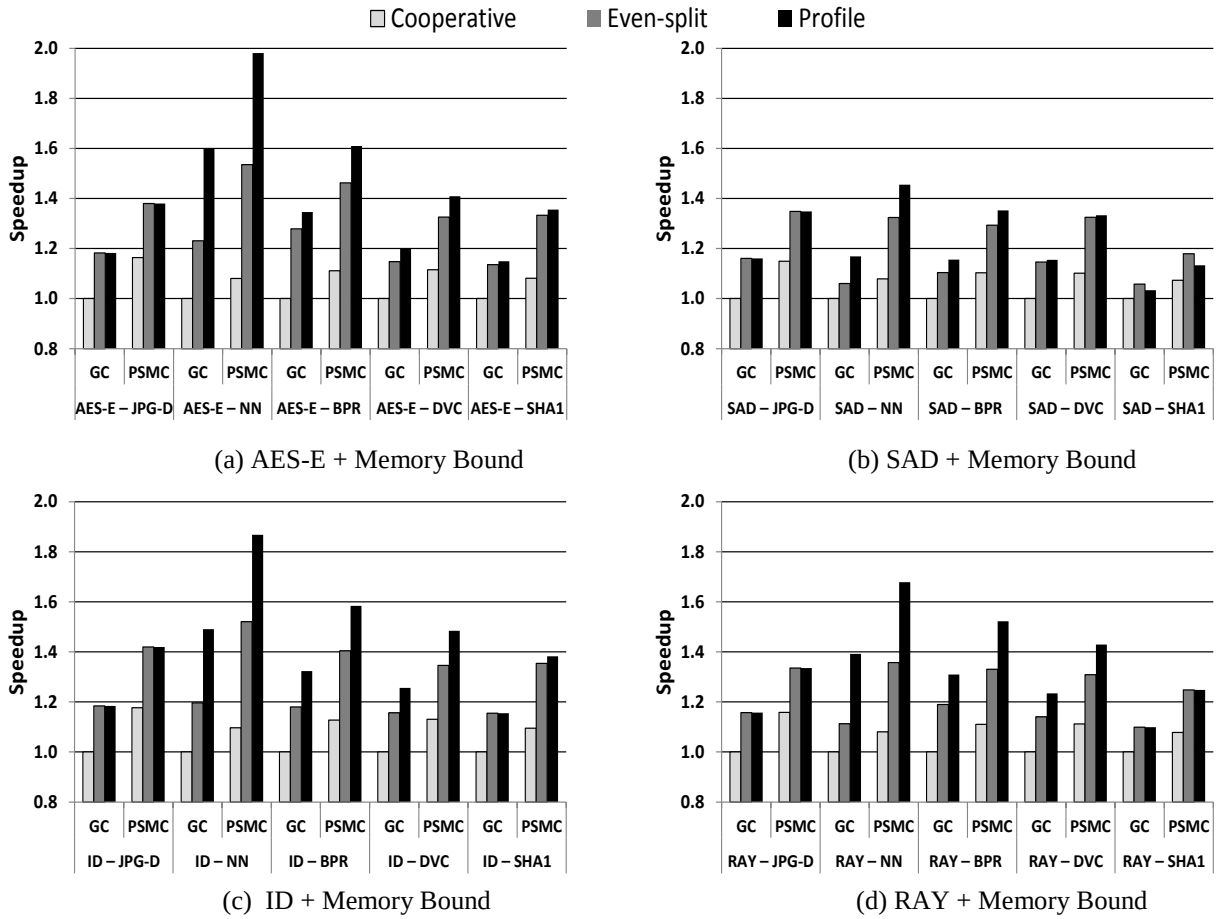
equal access to GPU resources. *Profile*, on the other hand, tries to assign the number of resources to each application that maximizes system throughput. The *best* partitioning method would depend on the system's and/or user's goals.

4.2.1 Performance Impact of Application Assignment Methods for Spatial-Multitasking GPUs Supporting PSMC

As shown before, different applications respond differently to frequency. I hypothesize that assigning more compute-bound applications to fast SMs and less compute-bound applications to slow SMs will result in higher overall performance. I compare the performance of the two assignment methods: (i) compute-bound applications to faster SMs and memory-bound applications to slower SMs (CFMS) and (ii) compute-bound applications to slower SMs and memory-bound applications to faster SMs (CSMF). I use *Profile* partitioning and I run several combinations of two applications for a time corresponding to 5M GPU cycles of the slowest SM. I use *Profile* because it provides better overall system performance. Table 3 shows the results, normalized to cooperative multitasking with GC; CFMS is better and never worse than CSMF. The difference (Δ) is greatest when a compute-bound application executes with a memory-bound one, and negligible for applications with similar characteristics. For the remainder of the paper, I use CFMS.

4.2.2 Impact of SM Partitioning Algorithms on Performance of Spatial-Multitasking GPUs Using CFMS

First, I run combinations of two applications on the GPU to study the performance improvement of using PSMC with CFSM and my partitioning algorithms over GC. In Figure 5 each graph shows the speedup of a compute-bound application running with the memory-bound ones. Assigning faster SMs to compute-bound application(s) and slower SMs to memory-bound application(s) leverages the PSMC benefits provided by WID. Figure 4 showed that AES-E and ID scale better with frequency than SAD or RAY. Hence, combinations of either AES-E or ID with memory-bound applications show higher speedups than when those memory-bound applications are combined with SAD or RAY. *Even-Split* partitioning outperforms cooperative multitasking, and *Profile* generally outperform *Even-Split*.



(a) AES-E + Memory Bound

(b) SAD + Memory Bound

(c) ID + Memory Bound

(d) RAY + Memory Bound

Figure 5: Speedup of compute- and memory-bound application combinations for spatial multitasking partitioning algorithms using both GC and PSMC. The results are normalized to cooperative multitasking with GC.

Figure 6 shows PSMC performance when two compute- or two memory-bound applications run simultaneously on the GPU. For brevity, I only present results for a few combinations. When two memory-bound applications co-execute, neither benefits much from higher frequency SMs. When two compute-bound applications co-execute, the benefit achieved by one application using fast SMs is largely offset by the penalty of the other using slow SMs.

Figure 7 shows the geometric means of the speedups of the application combinations shown previously. The first four pairs of columns are the geometric means of the combinations shown in Figure 5, the fifth are the means from Figure 6, and the sixth column shows all combinations. The most compute-intensive applications gain the most from PSMC and spatial multitasking when assigned high frequency SMs. The overall speedup using PSMC for the different partitioning algorithms ranges from 18%-20% over their counterparts using GC. Compared to cooperative multitasking with GC, these speedups increase to 33%-41% across all tested combinations, and 36%-46% for cases that combine compute- with memory-bound applications.

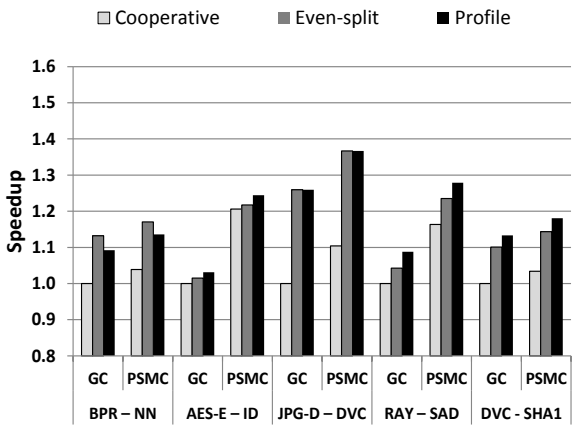


Figure 6: Speedup of combinations of two compute- and two memory-bound apps for different partitioning algorithms with GC and with Per-SMC. The results are normalized to cooperative multitasking with GC.

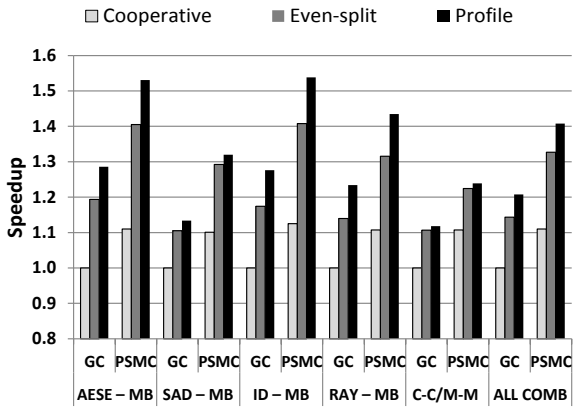


Figure 7: Geometric mean speedup of combinations of compute- and memory-bound applications, and the geometric mean of speedups for all combinations. Results are normalized to cooperative multitasking with GC.

4.2.3 Combinations of Three Applications

I extended the previous experiments to study combinations of three applications that might run together in a real scenario. For example, ID, JPG-D, and DVC together represent a user decoding and editing video and images. AES-E, DVC and NN show someone encrypting and decoding a high-definition video while searching for objects of interest in that video.

Figure 8 shows the speedups of the different partitioning algorithms over cooperative multitasking with GC. As expected, speedups are best when combining a mix of compute- and memory-bound applications. *Profile* results in better speedups than *Even-Split* for most

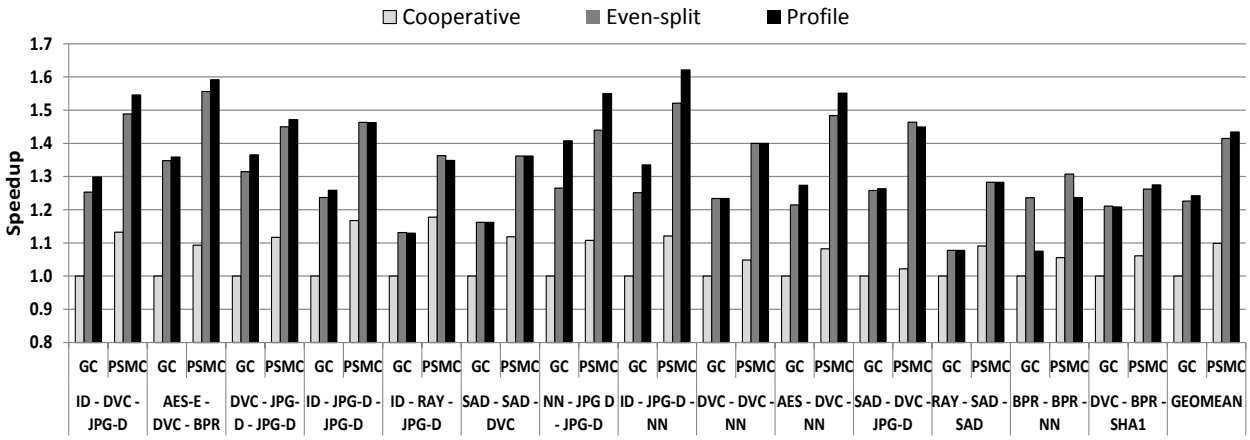


Figure 8: Speedups of combinations of three applications using different multitasking partitioning algorithms with GC and PSMC. The results are normalized to cooperative multitasking with GC.

combinations. For some combinations Figure 8 shows that *Profile* performs worse than *Even-Split*, which is initially an unexpected result. However, *Profile* uses information from running the applications in isolation; when multiple applications execute on the GPU, the contention between applications in the shared resources (such as interconnect and memory) can affect performance.

Resource partitioning algorithms with PSMC obtain average speedups of 19% for both *Even-Split* and *Profile*, compared to GC. Compared to cooperative multitasking with GC, speedups are 42% and 43% when running three applications with spatial multitasking and PSMC.

4.3 Summary

I propose a variation-aware assignment of SMs to applications in a GPU that supports spatial multitasking and PSMC. I characterize GPGPU applications as compute-bound or memory-bound based on their change in performance as the frequency of SMs is increased. Next, I apply a variation-aware assignment of SMs to applications that co-execute with spatial multitasking. Our proposed technique that assigns faster SMs to compute-bound applications and slower SMs to memory-bound applications demonstrates an improvement in performance. I also evaluate different SM partitioning algorithms that take advantage of PSMC in a multitasking GPU. When executing two or three applications concurrently, I achieve speedups of 18%-20% using *Even-Split* and *Profile* partitioning algorithms, respectively, compared to the same partitioning algorithms with global clocking. These speedups increase to 36%-46% when compared to the performance of cooperative multitasking with global clocking.

5 QoS-Aware Dynamic Resource Allocation for Spatial-Multitasking GPUs

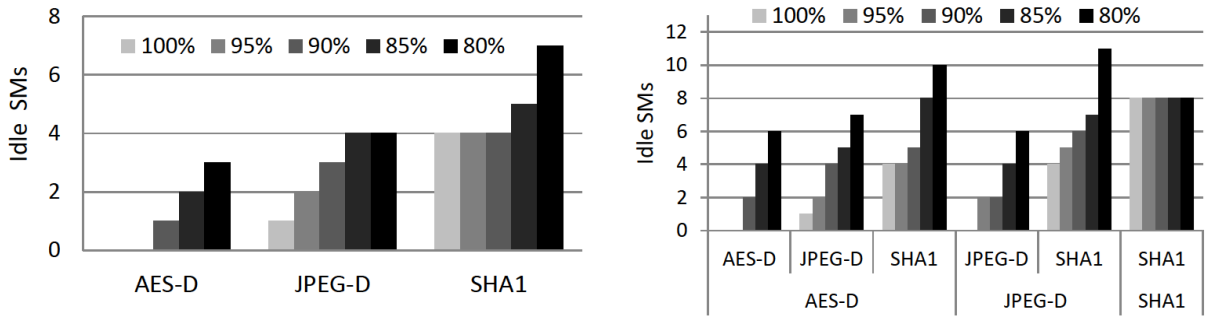
In this section I explore how spatial multitasking can improve system performance and power consumption when running quality of service (QoS) applications on a spatial multitasking GPU. For this work I also model a NVIDIA Quadro FX 5800 GPU using GPGPU-Sim.

5.1 Methodology

AES Decoding (AES-D), JPEG Decoding (JPEG-D) and SHA1 are our QoS applications; they present different characteristics and represent applications that may have a QoS requirement: Encryption algorithms need to keep pace with communication or media playback, and JPEG-D is required to decode frames at a certain rate. Image Denoising (ID), Ray Tracing (RAY), Dirac Video Codec (DVC), Sum of Absolute Differences (SAD), and Fractals are our *best-effort* applications.

I compare the power and performance implications of using spatial vs. cooperative multitasking while meeting the QoS requirements of different applications. I measure power differences by noting the number of SMs that must be enabled for each method and I measure performance differences in terms of the number of instructions completed during the same time span [59] for each method. For our experiments, I assume that the QoS applications meet their QoS using cooperative multitasking. I then evaluate if spatial multitasking can also meet these requirements, and if so, how many SMs can be left idle to save power or assigned to best-effort applications to overall system improve performance.

To establish a QoS requirement for each application, I measure the work (number of instructions) it completes when allocated 100% of the GPU resources (30 SMs) for 50% of the simulated execution time (*i.e.*, cooperative multitasking). This same instruction count must then



(a) (b)

Figure 9: (a) SMs not required by each QoS application running in isolation on the GPU. (b) SMs not required for two QoS applications sharing the GPU.

be met by a partial allocation of SMs for 100% of the simulated time. Different QoS requirements do not change the fundamental approach, merely the result of applying it. To test the effects of "tighter" vs. "looser" requirements, I also examine 95%, 90%, 85% and 80% of the calculated QoS.

To evaluate the benefits of spatial multitasking over cooperative multitasking for QoS GPGPU applications, I examine three execution scenarios. First, one QoS application is executing in isolation, and I determine the minimum number of SMs needed to satisfy its QoS. Second, I examine two concurrently-executing QoS applications, and determine how many SMs are not needed to meet the applications' QoS requirements—which depends in part on the co-executing applications. Last, I examine combinations of one best-effort and one QoS application to determine how allocating the "extra" SMs (those not needed by the QoS application) to the best-effort application can improve performance.

5.2 Profile-based Resource Allocation

In this section, I present application profile information showing how many SMs each QoS application needs to meet its requirement. First, I run each QoS application in isolation using up to 50% of the GPU resources (15 SMs) for 100% of simulated time and measure the number of

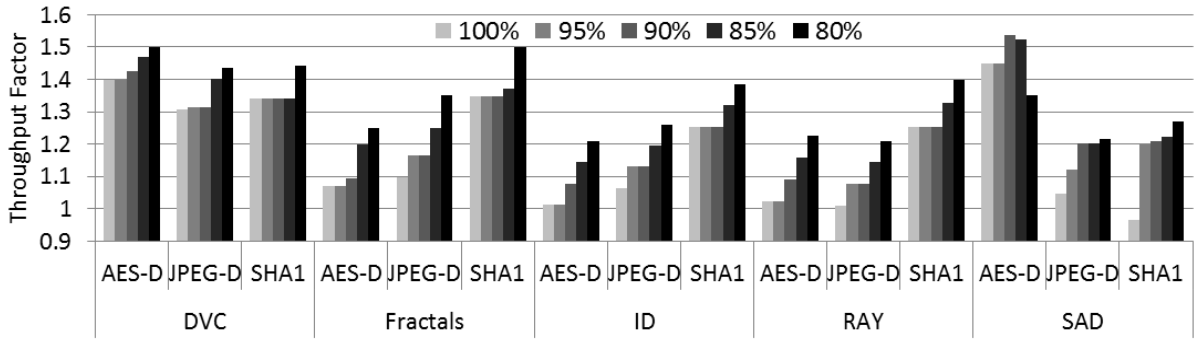


Figure 10: The throughput increase of best-effort applications when spatially-multitasked with one QoS application for 100% of the simulated time as compared to running in cooperative multitasking using all the resources (30 SMs) for 50% of the simulated time.

SMs (out of the 15) that can be disabled to reduce power consumption while meeting QoS requirements (Figure 9 (a)). I can disable 4 SMs for SHA1's 100% of execution time but none for AES-D at 100% of the calculated QoS. As the QoS requirement is relaxed, more SMs can be disabled.

Second, I concurrently run two QoS applications using spatial multitasking on the GPU, and evaluate the number of SMs that can be disabled while still meeting their QoS (Figure 9 (b)). Spatial multitasking allows the GPU to disable 11 SMs when concurrently executing SHA1 and JPEG-D; with cooperative multitasking there would be no idle SMs—each application uses 100% of the resources for 50% of the time.

Finally, I examine all possible SM allocations when one QoS and one best-effort application co-execute on the GPU. I determine, for each pairing, the minimum number of SMs for the QoS

Table 4. Average power savings when spatially-multitasking two QoS applications, if unneeded SMs are left idle vs. power-gated.

QoS	Idle SMs (W)	Power Gated SMs (W)
100%	2.6	6.9
95%	5.8	11.2
90%	8.8	15.8
85%	13	22.2
80%	18.1	28.6

application to satisfy its QoS requirement, then measure the performance improvement of the best-effort application when it is allocated the extra SMs. Figure 10 shows the throughput increase factor of best-effort applications when they are allocated the "extra" SMs with spatial multitasking. This is computed by dividing the amount of work that the best-effort application completes when spatially-multitasked with the QoS application, divided by the amount of work that the best-effort application completes when cooperatively-multitasked using all the resources (30 SMs) for half of the time. When a best-effort application such as Fractals, ID and RAY concurrently runs with SHA1, their performance increases at least 20% for the 100% QoS requirement. Cooperative multitasking, however, cannot allocate unused SMs to best-effort applications to increase performance.

When I only run QoS applications, any SMs unneeded for QoS is left idle; idle SMs contribute to power savings because they consume less dynamic power than active SMs. Current GPUs do not employ core power gating, but future ones could as a result of the increasing number of cores and the increasing importance of power consumption especially in embedded devices. Using GPUWattch [64], I determine the power consumption per SM for every application and then compute the power savings obtained by leaving SMs idle as well as by power gating the otherwise-idle SMs when running two QoS applications simultaneously (using spatial multitasking) on the GPU (Table 4).

5.3 Impact of Co-scheduled Applications on Performance

The previous section showed that the number of SMs required by QoS applications to meet their performance requirements with spatial multitasking depends on any co-executing applications. For each QoS application, I determined which best-effort application most

degraded the QoS application's performance. Table 5 reports this performance degradation for each QoS application relative to its performance in isolation with the same number of SMs. I show results for both 10 SMs (1/3 of the GPU) and 15 SMs (1/2 of the GPU). AES-D does not have notable performance loss, but JPEG-D and SHA1 are more sensitive to co-executing applications. Even with 50% (15) of the SMs, JPEG-D and SHA1 exhibit up to 16% and 18% performance loss (respectively) in the worst case.

It is thus difficult to statically determine QoS applications' required SM allocation. If the application(s) that might co-execute with it are not known a priori, I must conservatively allocate SMs to ensure the QoS is met for the worst case. This could considerably reduce or completely negate the performance benefit of spatial multitasking over cooperative multitasking. I further examine this problem in the next section, and propose a method to address it.

Table 5. Maximum performance loss of each QoS application when sharing the GPU with another application vs. alone with the same # of SMs

Application	**Max. performance loss (15 SMs)**	**Max. performance loss (10 SMs)**
AES-D	0.3%	0.3%
JPEG-D	16%	23.2%
SHA1	18.2%	24%

5.4 Dynamic Resource Allocation

As previously discussed in Section 5.3, an application's performance depends on any co-executing applications due to contention in shared resources such as memory and interconnect. Thus, I propose a runtime algorithm to estimate QoS application performance and dynamically allocate the required SMs to them.

For performance estimation, I assume that application performance increases linearly with the number of SMs. I assign an initial number of resources to the QoS application and measure its performance at that point; our performance estimation is the line that contains that point and a slope equal to the performance divided by the number of SMs at that point. I can plot this estimation on a graph of performance vs. SM allocation by drawing a line between the current measurement point (# SMs, performance) and a point at 1 SM and a performance level equal to the measured performance divided by the SM count for that performance. Most applications show sub-linear speedups with the number of allocated SMs. Hence, when used as a guide primarily for estimating a *reduction* in SMs, this performance estimation is generally pessimistic. Figure 11 shows the performance of SHA1 running in isolation for allocations of 1 to 30 SMs.

Since this estimation method is not 100% accurate, however, I use an iterative approach as follows:

1. I initially assign 50% of the SMs to a QoS application and measure its performance after it has run on those SMs for at least the duration of one thread block. I use this sample point to create a linear performance model.
2. If the current performance exceeds the QoS requirement, I determine whether or not the QoS

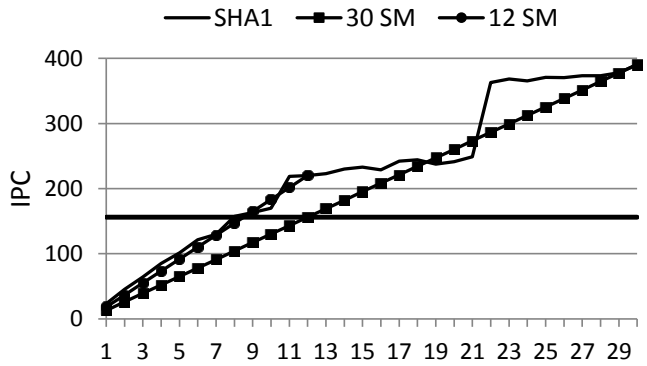


Figure 11: Linear approximation of SHA1, running in isolation on the GPU, from 30 SM for a target QoS of 160 IPC. X-axes are the number of SMs.

would still be satisfied with fewer SMs based on the model. If so, I decrease the allocated SMs as predicted by the linear model. On the other hand, if the QoS is not satisfied, I increase the SM allocation according to the model.

3. I re-evaluate the performance after running the application with the adjusted number of SMs, and repeat step two if appropriate.

To reallocate an SM from one application to another, I first need to make sure that the SM is no longer executing work. There are three ways to do this: (i) migrate the remaining work to a different SM; (ii) terminate the work and later restart it on another SM; or (iii) wait for the SM to finish executing. As each SM on a GPU has many threads executing simultaneously, each keeping their intermediate results in large register files and local memory, migrating work across SMs requires moving a large amount of state and thus incur a great deal of overhead. Terminating the work and later restarting it on another SM can be very costly because some threads perform a large amount of work that might then be lost. For this work, I choose the third option: wait for the SM to finish executing the remaining work it had been assigned from the first application; then, allocate the SM to the new application.

Figure 11 shows how our algorithm works. SHA1 needs to satisfy a QoS of 160 IPC, shown as a horizontal line. I start at 30 SMs, and after executing the application for the duration of a thread block, our algorithm estimates that 12 SMs are needed to satisfy the QoS. I wait for the extra SMs to finish execution of any remaining work, then I reduce the number of SMs to 12. After executing our application now on 12 SMs for the duration of a thread block, the algorithm runs again and estimates that only 9 SMs are needed. With 9 SMs the QoS is satisfied, and if I run the algorithm again it determines that I cannot further reduce the SM allocation.

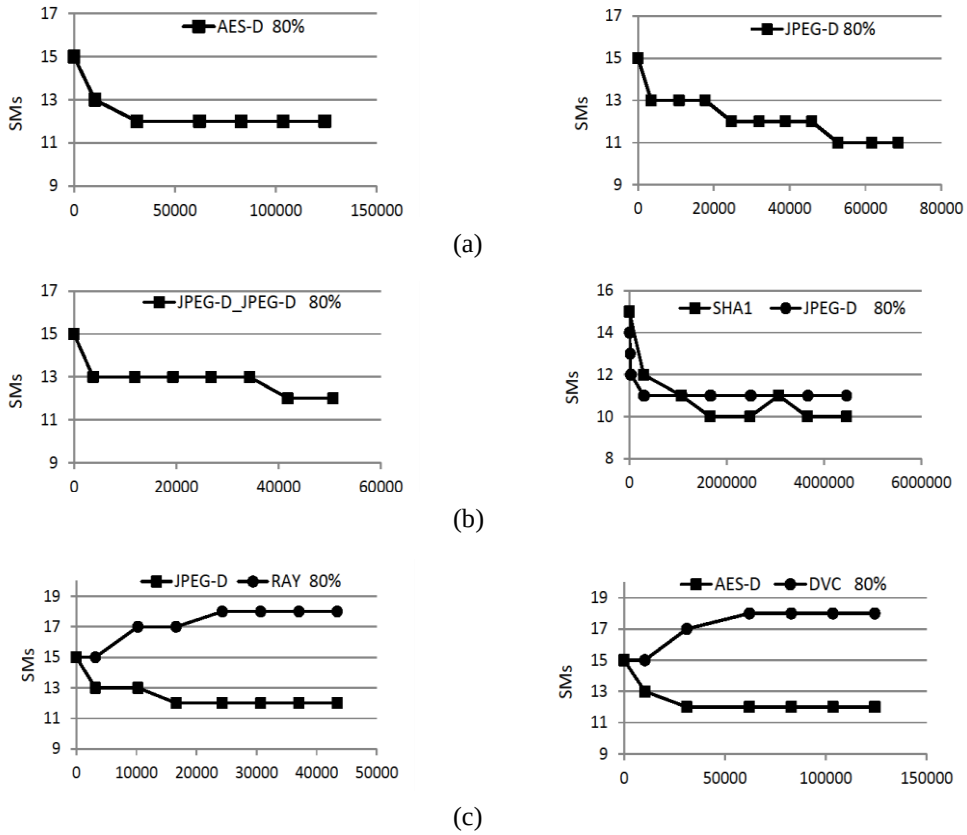


Figure 12: (a) Convergence of each QoS application, in isolation, to the minimum # of SMs to satisfy its QoS. (b) Convergence of two QoS applications spatially-multitasked on the GPU to the minimum # of SMs to satisfy their QoS. (c) Convergence of one QoS application to the minimum # of SMs to satisfy its QoS when spatially-multitasked with a best-effort application. The x-axes represent the # of GPU cycles (passage of time).

For our experiments, the main concern is to satisfy the QoS requirements. Thus, I conservatively choose the middle point of the line for our next SM allocation, intentionally attempting to over-estimate the required SM allocation. This requires more iterations, but reduces the probability of backtracking to a higher allocation (if the QoS is not met). Although I always want to satisfy the QoS requirements, the proposed approach may still violate QoS requirements for short time periods due to inaccurate estimations. However, such violations can be tolerable for soft QoS requirements, as long as the performance returns to an acceptable level.

Figure 12(a) shows how the number of SMs converges to the minimum over time, the extra SMs are left idle. The *x*-axes represent the cycle count. AES-D converges to 12 SMs and JPEG-

D to 11 SMs. Figure 12(b) shows how the number of SMs needed by two QoS applications running simultaneously using spatial multitasking converges to the minimum required to satisfy 80% QoS, with the unused SMs left idle. Two copies of JPEG-D running on the GPU need 12 SMs each, as opposed to the 11 SMs that would have been required for one copy of JPEG-D in isolation. When SHA1 and JPEG-D co-execute on the GPU, JPEG-D needs 12 SMs if SHA1 has more than 12 SMs, but when SHA1 is reduced to 12 SMs or less, JPEG-D can satisfy its QoS with 11 SMs. Figure 12(c) shows combinations of one QoS and one best-effort applications. As the number of SMs assigned to the QoS application decreases, the number of SMs assigned to the best-effort application increases.

I observe that all the QoS applications converge to the minimum number of SMs needed to satisfy their QoS. Each application takes a different number of cycles to converge, depending on the duration of their thread blocks. This is because I execute the application for a complete thread block before running our algorithm. If I decide to reassign an SM, I also wait for that SM to complete its in-progress work before re-allocating it. When running just QoS applications on the GPU, the SMs that are not needed by the QoS applications could be power gated during the search process of the algorithm. Since I start from a conservatively high number of SMs and reduce from there, I rarely suffer from overhead related to waking-up the power gated SMs during the (re-)allocation process.

Dynamic SM allocation could be done by the GPU hardware, which already assigns computations to resources (*e.g.*, thread blocks to SMs) without OS intervention. This is because many threads with short execution time run on the GPU, and frequent OS intervention would incur excessive overhead. For a spatial-multitasking GPU, its hardware would also be responsible for assigning thread blocks to the SMs and thus it tracks which SMs are allocated to

which applications. For QoS support, the OS now must inform the GPU of an application's performance target so the GPU allocates enough SMs to satisfy the application's QoS requirement.

5.5 Summary

Spatial multitasking on the GPU can satisfy the performance requirements of QoS applications by assigning them enough computing resources to satisfy its performance requirements, while allocating unused SMs to best-effort applications improving overall system performance or leaving them idle contributing towards power saving.

I show that the performance of the QoS application depends on what other applications run simultaneously on the GPU, this is due to contention on the shared resources such as interconnect and memory. This is the reason why a static allocation of resources to the applications is not a good approach. I show how a linear approximation algorithm allows us to dynamically allocate the minimum number of resources to the QoS application, so that it satisfies its performance requirements no matter what other applications run simultaneously on the GPU.

By using spatial multitasking with QoS applications on the GPU I obtain 7W power consumption reduction or 17.57% performance improvement for co-executing best-effort applications.

As the GPUs become parallel processors and support simultaneous execution of multiple applications, allocation of GPU resources to co-executing applications will become more relevant and should be further researched.

6 Resource Allocation and Fairness in a Spatial Multitasking GPU

General Purpose GPU (GPGPU) spatial multitasking requires partitioning the streaming multiprocessors (SMs) among simultaneously-running applications. But this partitioning can be performed based on a variety of goals, such as performance, power-savings, and others. If the aim is to maximize overall system throughput, but one co-executing application has significantly higher throughput than the other, the high-throughput application will receive nearly all SMs, penalizing the other application's performance relative to what it may have achieved. A possibly more fair allocation may be to evenly distribute the resources to the applications, but this approach may not be the most effective use of resources, penalizing overall throughput. In this thesis I propose and evaluate a variety of possible resource allocation policies, and determine their effectiveness at providing a balance of individual application and overall system performance.

6.1 Proposed Fairness Policies

As stated above, I examine several different possible resource allocation policies, each of which could, from some point of view, be considered "fair". These include:

- Equal compute resources for each application.
- Equal throughput for each application.
- Equal speedup for each application.
- Maximum system throughput such that applications are not slowed relative to cooperative multitasking.

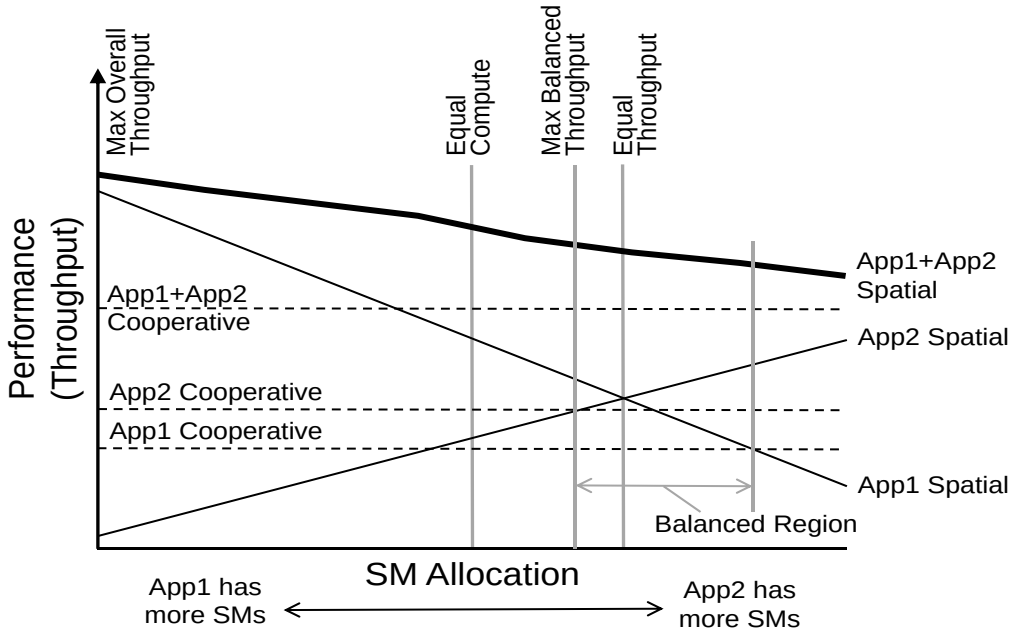


Figure 13: Example system and individual application throughput for different SM allocations using spatial multitasking (solid lines). Dashed lines show cooperative throughput for comparison. Vertical lines indicate the allocation required to achieve different fairness metrics.

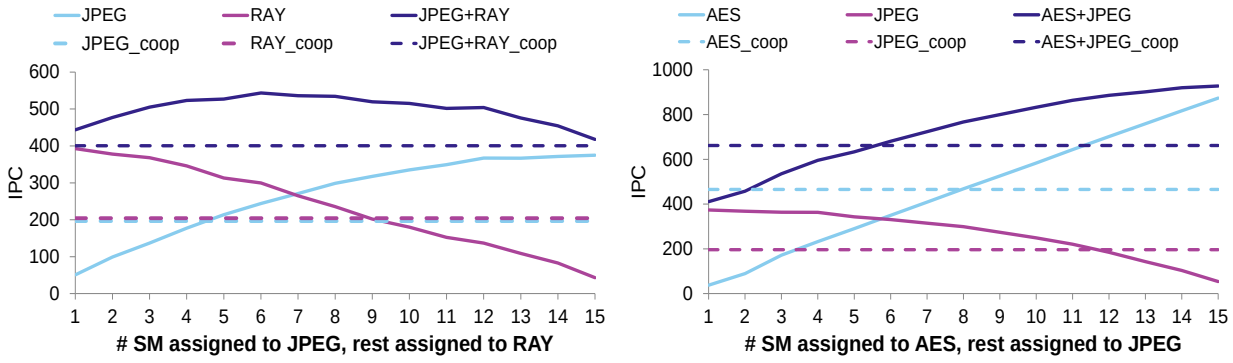
Figure 13 illustrates three of the fairness policies using two fictitious applications: *App1* and *App2*. Solid lines represent individual application performance when running concurrently on the GPU with spatial multitasking, for all possible resource allocations. When App1 gets most of the SMs it has high performance, whereas App2 gets few SMs (and vice versa). The dark line on top is the combined performance of both applications (system performance). The dashed lines show the performance of the applications running in isolation using cooperative multitasking (all resources for 50% of the time). The vertical lines are the allocations meeting the following fairness criteria: *Equal Compute*, *Max Fair Throughput*, and *Equal Throughput*.

6.2 Comparison of Fairness Metrics

In this section, I examine different combinations of applications, and, for each fairness goal described previously, determine the resource allocation that meets that goal. In the next section, I discuss how I might predict the desired allocation at runtime.

Figure 14 shows the performance curves for combinations a) JPEG+RAY and b) AES+JPEG. Each application's throughput generally increases as it is assigned an increasing number of SMs, though different applications improve at different rates. For comparison, the throughput achieved with cooperative multitasking is shown as horizontal lines in the figures. As previously stated, cooperative multitasking is the performance that the application obtains when running in isolation on the GPU, for half of the total simulated time.

If half of the resources (8 SMs) are assigned to each of JPEG and RAY (Figure 14(a)), JPEG benefits considerably compared to cooperative multitasking (52% improvement), while RAY performs 15% better. If I instead assign 7 SMs to JPEG and 9 SMs to RAY, both applications have almost equal throughput and both applications benefit from spatial multitasking (38-30% speedups). Assigning 6 SMs to JPEG and 10 SMs to RAY provides the highest overall system speedup (36% speedup compared to the cooperative case) and provides speedups to both applications with respect to the cooperative multitasking case. *Equal speedup* would be at the same point as *Equal Throughput* because their cooperative performance is similar.



(a) (b)

Figure 14: For (a) JPEG running with RAY and (b) JPEG running with AES, the individual application and system throughput for all the different SM allocations among two applications for spatial multitasking. Horizontal dashed lines show cooperative throughput for comparison.

Figure 14(b) shows the performance of spatially multitasking AES and JPEG. Assigning most of the SMs to AES results in the highest system throughput, starving JPEG off. Assigning half of the resources to each application provides JPEG with 52% speedup over the cooperative case, but no benefit for AES. *Equal Throughput* is obtained when AES gets 6 SMs and JPEG 10 SMs, slowing down AES. Assigning 11 SMs to AES and 5 SMs to JPEG maximizes system performance while each application performs better than cooperative multitasking. *Equal Speedup* is obtained when AES gets10 SMs and JPEG gets 6 SMs.

Table 6 shows these results along with other combinations of applications. I see that different fairness policies mean different tradeoffs between applications' and overall system performance. For AES+JPEG and ID+SHA1, *Equal Throughput* slows down one of the applications to match its throughput to the other one, resulting in lower system performance. *Equal Speedup* results in a good overall system performance with no application being considerably slowed down. For JPEG+RAY, *Max Fair Throughput* has the same allocation as *Max Unfair Throughput*, but for some other combinations of applications (AES+JPEG and RAY+SHA1) the allocations differ and *Max Unfair* starves one application. I observe similar results from running three applications, *Equal Throughput* does not provide a good overall system performance, *Equal Speedup* and *Max Fair Throughput* provide higher overall system throughput and better individual application performance. *Max Unfair Throughput* also starves one of the applications.

Table 6. For an illustrative subset of tested application combinations, the number of SMs assigned by each fairness policy and the resulting raw (T)hroughput in IPC and (S)peedup relative to executing the same application set with cooperative multitasking. The highest speedup in each row is highlighted.

	Equal Compute Resources			**Equal Throughput**			**Equal Speedup**			**Max Fair Throughput**			**Max (Unfair) Throughput**		
APP.	SM	T	S	SM	T	S	SM	T	S	SM	T	S	SM	T	S
JPEG	8	299	1.52	7	270	1.38	7	270	1.38	6	244	1.24	6	244	1.24
RAY	8	235	1.15	9	265	1.30	9	265	1.30	10	300	1.48	10	300	1.48
JPEG+ RAY	16	534	1.33	16	535	1.34	16	535	1.34	16	544	1.36	16	544	1.36
AES	8	468	1.00	6	349	0.75	10	583	1.25	11	643	1.38	15	874	1.88
JPEG	8	299	1.52	10	331	1.69	6	249	1.27	5	221	1.12	1	54	0.27
AES+JPEG	16	767	1.16	16	680	1.03	16	832	1.26	16	864	1.31	16	927	1.40
ID	8	393	1.00	5	246	0.63	9	442	1.13	9	442	1.13	15	734	1.87
SHA1	8	169	1.13	11	225	1.50	7	152	1.02	7	152	1.02	1	22	0.15
ID+SHA1	16	563	1.04	16	471	0.87	16	594	1.09	16	594	1.09	16	756	1.40
PATH	8	412	1.00	1	52	0.13	11	555	1.40	11	555	1.40	15	756	1.9
SC	8	39	1.6	15	45	1.87	5	30	1.23	5	30	1.23	1	6.26	0.26
PATH+SC	16	450	1.1	16	97	0.23	16	585	1.37	16	585	1.37	16	763	1.8
JPEG	5	220	1.68	3	141	1.10	4	184	1.40	5	220	1.68	5	218	1.66
RAY	5	161	1.13	5	161	1.13	6	188	1.32	6	188	1.32	10	303	2.12
SHA1	6	120	1.19	8	160	1.59	6	123	1.22	5	101	1.00	1	17	0.17
JPEG+RA Y+SHA1	16	501	1.34	16	462	1.24	16	495	1.32	16	509	1.36	16	538	1.43
AES	5	295	0.95	3	177	0.57	5	295	0.95	5	295	0.95	14	817	2.63
ID	5	246	0.95	4	197	0.75	6	295	1.13	6	295	1.13	1	49	0.19
SHA1	6	127	1.26	9	184	1.82	5	106	1.05	5	106	1.05	1	21	0.20
AES+ID+ SHA1	16	668	1	16	558	0.82	16	696	1.04	16	696	1.04	16	887	1.32
RAY	5	161	1.13	6	188	1.32	5	161	1.13	4	143	1	1	51	0.36
RAY	5	161	1.13	6	188	1.32	5	161	1.13	4	143	1	1	51	0.36
ID	6	295	1.13	4	197	0.75	6	295	1.13	8	393	1.5	14	686	2.62
RAY+RAY + ID	16	617	1.13	16	573	1.05	16	617	1.13	16	679	1.24	16	788	1.44

(b)

Figure 15Figure 15 shows aggregated data for all the possible combinations of two applications for the different metrics. Figure 15(a) shows the speedup of the **least** "sped-up" application (light) and overall system speedup (dark) compared to cooperative multitasking. I use the geometric mean to aggregate these values. I see that, although *Max Unfair Throughput* results in the largest overall throughput, it has the greatest negative impact on the least-

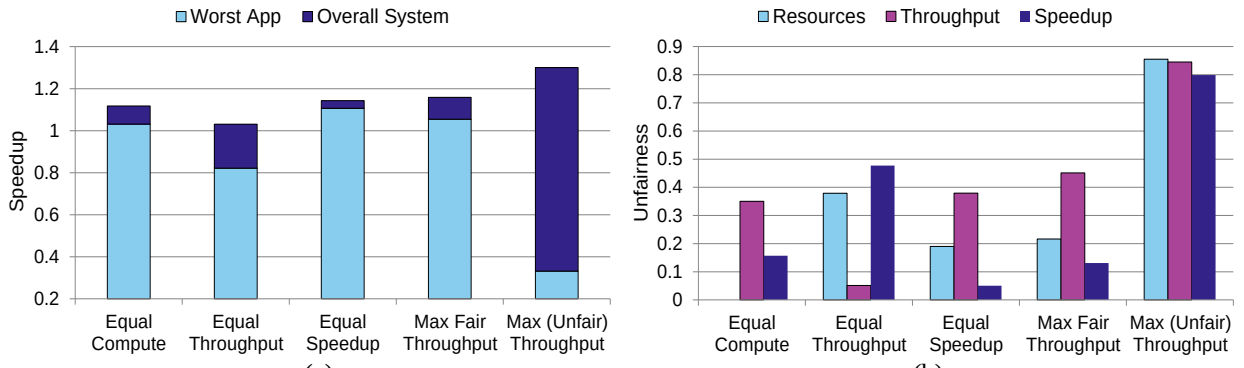


Figure 15: Speedup and Fairness information for all possible combinations of two applications for the different fairness metrics, a) shows speedup of the application that benefits least from spatial multitasking with respect to the overall system speedup and b) shows the degree of unfairness in the allocation of resources, throughput and speedup that the applications experience.

accelerated application. In fact, it significantly *decelerates* that application relative to cooperative multitasking. Several criteria, including *Equal Speedup* and *Max Fair Throughput*, exceed the performance of cooperative multitasking for each application and the system. *Max Fair Throughput* has higher system throughput keeping the speedup of the least-accelerated application above, but close to, 1.

Figure 15(b) shows unfairness by graphing aggregated *unfairness* for several possible metrics for each fairness criteria. I measure the unfairness of these metrics as the normalized difference between the highest and the lowest value from the two applications. For example, when measuring unfairness in the speedups of the applications I compute the normalized difference between the speedups of the most-accelerated and least-accelerated applications using the following equation:

$$(S_{max} - S_{mins}) / S_{max} \qquad (1)$$

where S_{max} is the largest speedup and S_{min} is the smallest speedup of the co-executing applications. Each fairness criteria that aims to equalize a particular metric (*Equal Compute*,

Equal Throughput, and *Equal Speedup*) achieves a very low degree of unfairness in that metric, though often at the expense of the other potential metrics. *Max Unfair Throughput* is significantly *unfair* in all metrics, since it targets performance and not fairness. *Max Fair Throughput*, on the other hand, is relatively fair in both resources and speedup—the unfairness of the throughput is due to different applications having different throughputs per SM, so achieving *similar speedups* does not achieve *similar throughputs*. Regardless, its throughput unfairness is significantly lower than that of *Max Unfair*, which starves the application with lower throughput.

6.3 Targeting Fair Allocation When Dynamically Allocating Resources

A system may execute a wide variety of GPGPU applications, and different combinations of them at different times. This means that it may not be reasonable to examine all possible SM partitions in advance for all possible combinations of applications to choose the allocation that best meets a particular performance and/or fairness goal. This section describes a method to use performance profiles of each application when they execute in isolation—specifically, the throughput the application achieves for each possible SM allocation in the absence of other applications—to estimate how the applications will perform when they execute together. I use these estimates to choose SM partitions at runtime, and then apply a linear approximation method and runtime performance monitoring to adjust this allocation to maximize the desired fairness metric.

6.3.1 Profile Based Allocation

I profile each application in isolation, for each possible number of SMs it could be allocated. For the targeted fairness criteria, I use the isolation data for each application to estimate the allocation of SMs that would meet that fairness criteria. In all of our test cases, I found that the

predicted SM allocation matched the allocation found through brute-force searching in the prior section. This means that, for at least some applications, the isolation profiling data can be used to choose a partitioning of the SMs at runtime, for the currently-executing combination of applications.

In other cases, however, performance may be dependent on the data size. If so, the application can be profiled in advance for several different data sizes. If the data size matches one of the profiles, I can use that profile. Otherwise, I can interpolate between the profiles to estimate performance [65]. The content of the data itself could also affect performance in some cases, though this is generally much less common for the type of highly data-parallel applications that are likely to target GPGPU execution. In that case, I could either restrict myself to a fairness criteria that requires no prior profile information, such as *Equal Compute*, or profile a variety of data sets in advance and attempt, at runtime, to determine which profile best matches the observed behavior of the application with the current data set.

In the next section, I discuss a method to adjust the allocation for cases where the isolation profile information may not as accurately reflect the run-time behavior. This loss of accuracy can be the result of contention between applications in shared resources (*e.g.*, interconnect and the memory system) that causes performance to scale differently from the isolation profile data. This adjustment method can also be used to accommodate staggered kernel execution, where some kernels may begin while others are already in progress, and some complete while others have not yet finished execution. This adjustment method can also be used to calculate the best partition of resources when I have no isolation profile data.

6.3.2 Dynamic Allocation of Resources Based on Execution Measures

As stated above, there may be cases in which an application's isolation data does not accurately predict its runtime behavior. In this case, the system uses the profile information to create an initial allocation that is then adjusted based on run-time performance monitoring and a linear approximation algorithm (similar to one that was previously-proposed) if it is not meeting the desired fairness criteria. Specifically, if the run-time behavior for any application deviates by more than an estimated SM's worth of performance from the isolation prediction, the allocation is adjusted using an updated estimate, based on runtime profile data, of how performance scales with SM allocation. For the case when I have not profile isolation data an equal allocation of resources can be used as a starting point.

The algorithm presented in Section 5.4 is used to allocate the minimum number of resources that a QoS application needs to satisfy its performance requirements, not considering fairness. This is done by first splitting all resources among the QoS applications, then progressively reducing the allocation to each if/when possible. I use a modified version of this algorithm to achieve a different goal: achieving a fair (according to the chosen fairness metric) partition of resources among applications executing concurrently on the GPU.

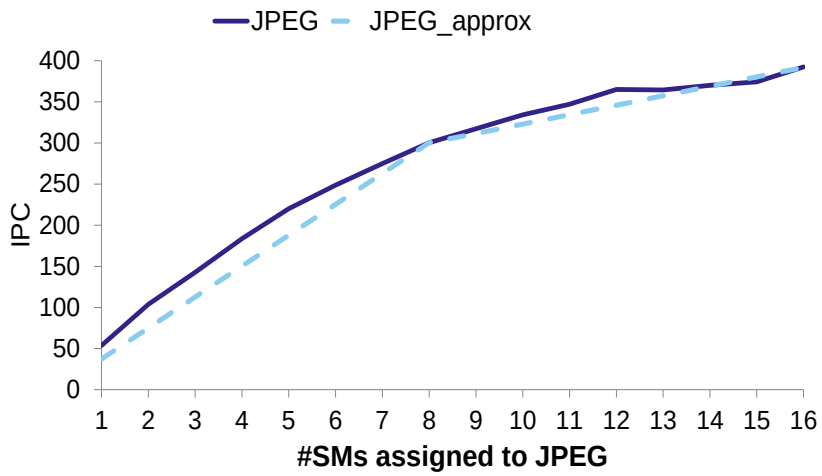


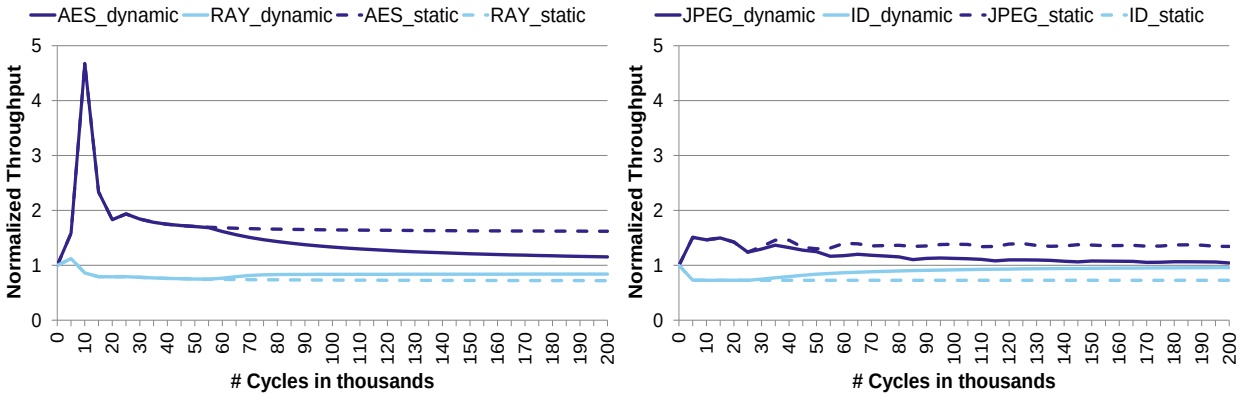
Figure 16: Graph of actual (solid line) and estimated (dashed line) JPEG throughput. Estimations are made from actual profiled performance with an initial execution on 8 SMs. The two different line slopes reflect different estimations for increasing vs. decreasing allocation.

Each application's performance is sampled by averaging it over a number of cycles equal to the duration of the longest thread block of all currently-executing applications. Some applications have particularly long-running thread blocks, which could decrease responsiveness, so limit the time to 25K GPU cycles, which I found to be effective in our experimentation.

I then estimate the performance for different SM allocations using two different linear functions of performance vs. SM count. For the case where the allocation increases, I use a linear function between the current (run-time profiling) datapoint to the application's isolation performance with all SMs. This latter point represents the extreme case where the application is, in fact, allocated all of the SMs. This method is based on the fact that, as an application's SM allocation increases, it should experience less contention from competing applications that now have fewer SMs on which to execute. For the case where the allocation decreases, I use a linear function that decreases from the current data point with a slope equal to the current performance divided by the current SM allocation. Figure 16 shows the two lines used to predict the performance of the JPEG application based on a performance measurement with an initial allocation of 8 SMs. The graph also shows the isolation performance for each possible SM.

After predicting each application's performance for different SM allocations, I choose the SM partitioning that is predicted to best satisfy our chosen fairness metric. To migrate SMs from one application to another, I first need the resources to become idle (SM Draining). A reallocated SM is allowed to complete its existing work, but no additional work can be assigned to it until reallocation is complete. The downside of this method is that the delay of SM Draining depends on the length of the application's thread blocks. Since kernels generally execute many times, and execution usually lasts much longer than a single thread block within the kernel, this overhead is usually reasonable. If this is not the case for some workload, the system could instead decide to not re-allocate the SMs as calculated.

I use our simulation platform to apply our dynamic linear approximation method to all combinations of two applications, using an equal allocation of SMs as a starting point. In all cases, this method is able, for my applications, to correctly determine the required allocation to meet the desired fairness metric with two passes of performance measurement followed by SM



(a) (b)

Figure 17: Convergence and overhead of the dynamic algorithm for two cases, a) *Equal Throughput* for AES and RAY and b) *Max Fair Throughput* for JPEG and ID. The graphs show cumulative application throughput normalized to the cumulative throughput achieved at the same cycle count if the "best" allocation for the targeted fairness metric is known from the start. Values close to 1 are better in terms of meeting the chosen fairness metric. Dashed lines show normalized throughput for a static allocation of equal SMs, which is also used as the starting point for the dynamic algorithm.

allocation adjustment.

Figure 17 shows the convergence of the dynamic approach, and illustrates the overhead of the dynamic method. I simulate each test case using the dynamic reallocation approach where each application in a test case is initially allocated half (8) of the SMs, and SMs are re-allocated during execution using our proposed method. I measure total achieved performance for each application every 5K cycles. I also simulate the same test cases for two static allocation cases: the "ideal" allocation for the chosen fairness metric (the allocation eventually achieved by the dynamic algorithm), and one where equal SMs were allocated to the applications for the entire simulation (the allocation used for the start of the dynamic algorithm). The graphs show application performance (throughput) normalized to the ideal static case at each sampled point in time. Values above 1 represent improved performance from an "excessive" SM allocation (tempered by SM draining overhead after dynamic reallocation); values below 1 represent penalized performance from "insufficient" SM allocation.

Figure 17(a) illustrates co-executing AES and RAY, targeting *Equal Throughput*. Each application was initially allocated 8 SMs, but the required allocation for *Equal Throughput* is 5 SMs for AES and 8 SMs for RAY. Early in the simulation, after some initial noise in the graph due to startup behavior, AES has a higher throughput than the ideal due to the increased SM allocation, whereas RAY has a lower relative throughput due to insufficient SMs. At approximately 50K cycles, three SMs are reallocated from AES to RAY, causing the normalized performance of each to more closely approach the ideal; performance continues to converge towards the ideal the longer the applications execute. The overhead of draining does not completely mitigate the advantage AES has from the initial SM allocation being higher than the desired allocation. The combination of a low initial SM allocation and the overhead of SM

draining prevents RAY from quite achieving the ideal performance. Based on this data, both applications would achieve performance within 5% of the ideal within 430K cycles.

Figure 17(b) illustrates co-executing JPEG and ID, targeting Max Fair Throughput. For the dynamic case, they are initially allocated 8 SMs each, but the ideal allocation for this metric is 5 SMs for JPEG and 11 for ID. This occurs in two steps, with 2 SMs reallocated at 24K cycles, and one additional reallocated at 35K cycles. In this case the dynamic performance also converges quickly towards the ideal, with both applications achieving within 5% of the ideal after 170K cycles.

6.4 Summary

Spatial multitasking can improve overall system performance and resource utilization compared to cooperative multitasking. But one of the main challenges it presents is how to divide the computing resources among the concurrent executing applications. I show that maximizing system performance can come at a significant cost of fairness of individual application performance. This thesis demonstrates that I can define a fairness metric that results in a partition of the SMs between simultaneously executing applications which increases performance relative to cooperative multitasking, and does not sacrifice individual applications performance. I further present a methodology to determine at runtime how to allocate GPU resources to meet this metric, and adjust that allocation if isolation-based performance predictions are determined to be inaccurate in the face of multiple co-executing applications.

7 Fine Task Migration for Graph Traversal Algorithms in Processing in Memory Architectures

In this Chapter, I study the feasibility and potential benefit of fine-grained work migration to reduce remote memory accesses in systems with multiple memory devices with Processing in Memory (PIM) capabilities. First, I propose a queuing framework to implement fine-grained migration; using this framework, vertex IDs are sent to the PIM stacks that will process them. Second, I propose hardware mechanisms that take advantage of PIM's proximity to memory to implement efficient queues and reduce the overhead of work migration. Third, I develop a high level timing model for the queueing framework to evaluate the proposed hardware support for efficient queues and to study the performance of work migration vs. a baseline where vertices are enqueued in a round robin fashion to the PIM stacks and remote memory accesses are performed as necessary. I propose and explore the potential benefits of a variety of task migration strategies and study their trade-offs under a variety of system configurations using this high-level model.

7.1 Motivation

To illustrate the number of remote vertices that are visited in an example graph algorithm, I observe the behavior of Breadth-First Search (BFS) that is a widely used algorithm in graph processing; it traverses a graph by starting at a root vertex and processing all its direct neighbors first before proceeding to process the next level of neighbors. BFS processes all the vertices that are at the same distance from the root before proceeding to process any vertices located one level further. Although the following data is taken from BFS, many other graph algorithms exhibit similar characteristics.

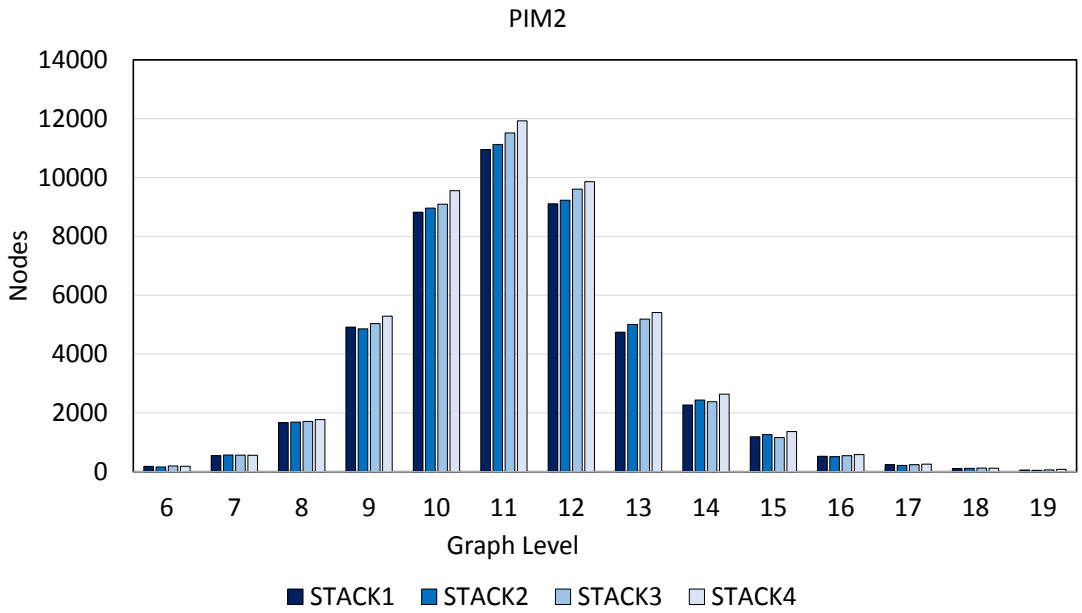


Figure 18. Neighbor distribution of a BFS traversal for a graph that has been partitioned among four PIM stacks. Graph level is the distance from the root vertex. This figure shows the spatial distribution of the neighbors for the vertices at each level of the graph that are located on PIM2.

I study a Google web graph taken from the Stanford Network Analysis Project (SNAP) [66], where the vertices are different webpages and the edges show the links connecting these webpages. This graph consists of 875,713 vertices and 5,105,039 edges. I evenly partition the graph data structure across a system with four PIM devices[1], PIM1 through PIM4. I perform a vertex-based partitioning, where I place equal numbers of vertices in each of the memory modules. All the outgoing edges of a vertex are placed on the same PIM as the vertex. This partitioning is corresponds to the one shown by Buluç and Maduri [53], called 1D Partitioning.

[1]While the specific graph partitioning used affects the data locality characteristics, my interest is in understanding the effectiveness of task migration independent of the graph partitioning scheme (*i.e.*, not all graphs can be effectively partitioned). Therefore, I use a simple partitioning here. Further, I believe effective task migration can avoid the high overheads typically required in graph partitioning schemes that seek to minimize communication.

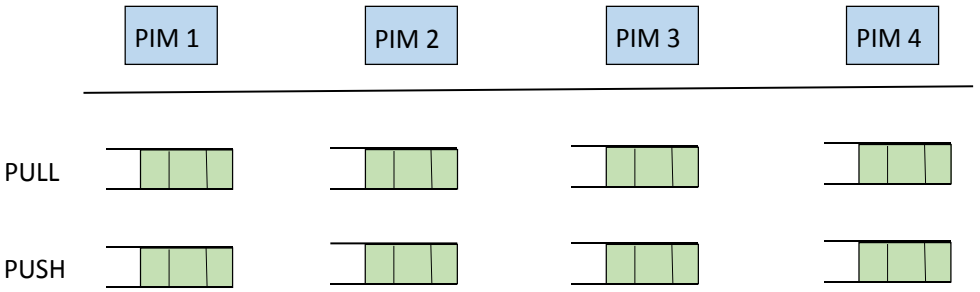


Figure 19. Queuing mechanisms used in my system to implement the graph algorithms and the work migration mechanisms.

Using this partitioning only the PIM where a vertex has been placed stores information related to that vertex. Figure 18 shows the neighbor distribution for the vertices located in a single PIM device (I arbitrarily select PIM2) at each level of the graph. The vertices that fall in PIM2 are the only neighbors that are local to PIM2. Neighboring vertices located in PIM1, PIM3, and PIM4 are remote to PIM2. There is an approximately uniform distribution of neighboring vertices among all the PIM stacks for this graph, resulting in the majority of the neighboring vertices being located in remote PIM devices. Processing such a large number of remote vertices implies many remote memory accesses, which can hurt performance and negate the benefits of using PIM.

7.2 System Architecture

In this section I first describe the system that I model in my experiments, and later I propose hardware support for efficient implementation of shared data structures in the system of study.

7.2.1 System Organization

Figure 2 shows the system of study, which consists of a host processor connected to multiple memory modules. Each memory module contains one or more memory dies and computation capabilities on a separate logic die connected through 3D stacking with the memory dies. In such a system, memory intensive computations can be offloaded to the PIM logic. I assume inter-PIM

links that allow each PIM to access any memory in the system, although it is always lower performance and higher energy to access remote memory. The study system architecture supports a shared, unified virtual memory address space among the host and PIM devices. Further, caches are kept coherent between the host and each PIM device as well as among the PIM devices.

The study system implements a CPU as our in-memory processor in each of the memory stacks, which enables the execution of general purpose programs. However, my evaluation methodology supports exploring the impact of PIM processors with greater degrees of parallelism. For my experiments, I also vary the number of PIM devices from 4 to 16. For my studies I don't use the host for the computation, all the computation are performed and evaluated in the in-memory processors. The host would be in charge of performing the graph paritioning and launching of the PIM computation, but I do not account for that overhead.

7.2.2 Queueing Framework

The queuing framework that I use consists of two queues per PIM device. One queue (PULL) is used to read the active vertices (vertices that require processing in the current iteration of the algorithm) and the other queue (PUSH) is used to keep track of the vertices that become active while processing the current vertices; those neighboring vertices will be processed in the next iteration of the algorithm. Figure 19 shows the queuing framework. These are actual queues that are in memory, no dedicated hardware is used for them for the storage, but some hardware support is provided to efficiently implement them as will be shown in Section 7.2.3. In work migration, when a vertex is processed, its neighbors are enqueued to the PIM device where their data is located, based on the partitioning of the graph data structure. In work migration vertices will be processed locally. While for the *baseline*, when a vertex is processed, its neighbors are

enqueued in a round robin fashion to the PIM devices in the system. Some vertices will be processed locally and some remotely. For both approaches some queue operations will be performed to local memory (local queue) and others to remote memory (remote queue). This framework is based on an implementation of BFS for distributed systems [53].

The queues are an array-based implementation. As there might be multiple threads simultaneously operating on the same queue, I use atomics to guarantee isolation and correctness. Currently I do not impose any restrictions on the size of the queues.

7.2.3 Hardware Support for Efficient Shared Data Structures

As the queues in the framework are shared and can be accessed by all the threads in the system, atomic instructions are used to update the index that threads need to use to access the array-based queues. The use of atomics results in performance overheads and increased contention.

I propose to leverage the proximity of PIM to memory to implement hardware support that can serialize the queue operations and guarantee atomicity without the need for explicit software atomics, thereby enhancing the performance of shared queues. The proposed work migration scheme benefits directly from these hardware mechanisms while other applications and execution models may also benefit from having more efficient shared data structures.

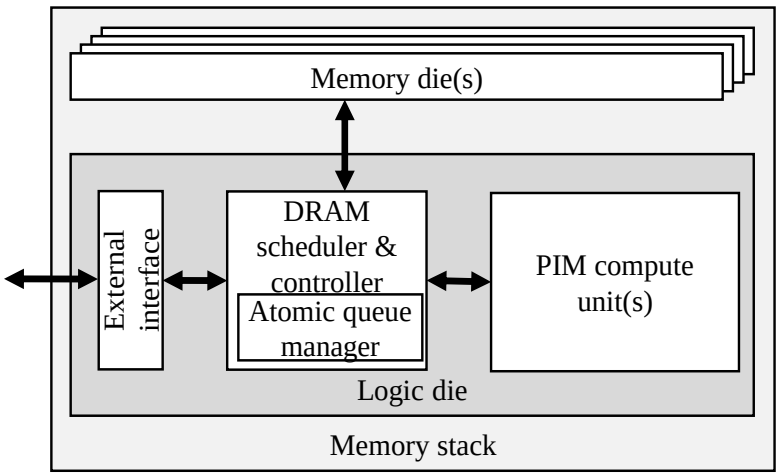


Figure 20. Hardware support for the atomic queues implemented in the memory controller.

Although the queues are hardware-accelerated, there are no dedicated storage resources for holding queue data, metadata, or queue pointers (except as an optional cache for performance optimization). Therefore, the scalability is not constrained by any hardware resources other than memory capacity itself. Further, this lack of dedicated state eases context switching and context management.

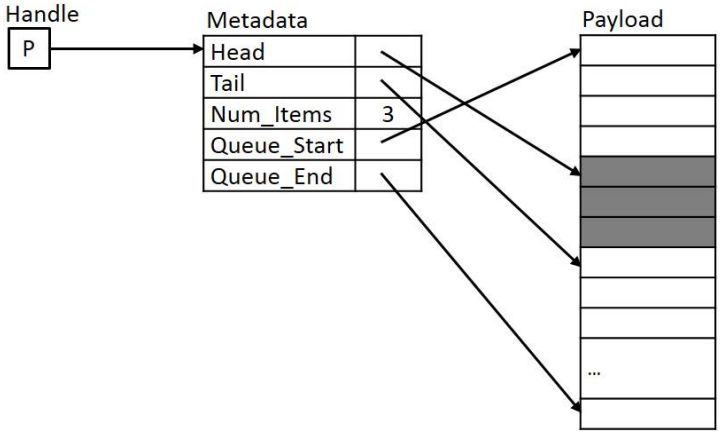


Figure 21. Metadata for a queue with three data elements in it.

Hardware support for such *atomic queues* is implemented in the PIM dies, at the DRAM controllers as shown in Figure 20. Each atomic queue is allocated such that it does not span

multiple memory modules or multiple memory channels (*i.e.*, all accesses to an atomic queue go through the same DRAM controller).

Before being used, each queue must be "registered" via a system call. The system call takes one or two arguments: the maximum size of the queue and, optionally, which memory module the queue must be placed on. If the second argument is omitted, system software selects a module to place the queue in. The queue registration allocates memory for the queue and sets up the necessary metadata. The registration call returns the following 2 values:

- P: an opaque handle to the allocated queue
- S: registration status, which returns a `SUCCESS` code if the queue registration succeeded or an error code if failed (*e.g.*, due to insufficient memory to allocate the queue)

The registration call also sets up the necessary metadata in a data structure located at the memory pointed to by P. For a queue the metadata includes head and tail pointers, a count of elements in the queue, and the start and end points of the memory space allocated for payload[2] storage of the queue and/or the maximum size of the queue. An example of a queue with three elements is shown in Figure 21.

Atomic queues are word-aligned and each enqueue or dequeue atomically inserts or removes a single-word entry to or from the queue. This capability is sufficient to enqueue vertex IDs or

[2] Throughout this document, "payload" refers to the data (to be) stored in the queue or other data structure implemented using the invention.

identifiers used in task queues. An enqueue operation is issued as a special store operation to the address of the queue (*i.e.*, the address of the metadata block returned at queue allocation) and the vertex ID as the data to enqueue. A dequeue operation is issued as a load instruction to the queue address. Specialized hardware at the DRAM controllers is responsible for updating the queue metadata as necessary as well as atomically performing the requested queue operation. Atomic queue metadata is allocated in a part of the addresses space that cannot be cached by the PIM or host processors to simplify the implementation. Note that this does not lead to performance overheads as the queue metadata are not manipulated by the processor after allocation. Further, to optimize the performance, the queue management hardware may contain a cache exclusively for storing recently used queue metadata (as all accesses to the queue must go through this hardware, its cache has no coherence requirements).

To enqueue a word to the queue, the application performs the following operation:

```
status = atomic_enqueue(P, D)
```

Here, `P` is the opaque queue handle returned when the queue was registered and `D` is the data payload word to be enqueued. If the operation succeeds, a SUCCESS code is stored in `status` upon completion. If the operation fails (*e.g.*, the queue is full), an error code is stored in `status`.

The preferred implementation is for atomic_enqueue() to be implemented as a single instruction in the host processor. An example format for such an instruction is as follows:

```
aeq rs1, rs2, rd
```

Here, the `aeq` mnemonic identifies the atomic enqueue instruction and `rs1` and `rs2` identify the two source operands: a register containing the queue handle and a register containing the data word to be enqueued (`P` and `D` in the above example, respectively). Note that alternative implementations may allow either or both of the operands to be specified as constants encoded in the instruction or to be computed using registers and/or constants (*e.g.*, the queue handle formed by adding a constant operand from the instruction to a value in a register). The register `rd` specifies where to store the status of the operation.

When executed on the host processor, the `aeq` instruction generates an "enqueue" operation to the memory module to which the address pointed by handle `P` maps to. Parameters `P` and `D` are passed as input operands to the enqueue memory operation (much like the address and data operands of a store operation). Upon receiving the enqueue(`P`, `D`) operation, the near-memory processor performs the following operations atomically with respect to other queue operations using the same handle `P`:

- Uses the first operand (`P`, in this case) to access the queue metadata structure.
- If the metadata indicates the queue is full:
    - Does not change anything in the queue and returns an error code to be stored in the status register at the host (`status`).
- Otherwise (*i.e.*, if the queue is not full):
    - Adds the data word (`D`, in this case) to the next available slot of the data payload storage of the queue
    - Update queue metadata as appropriate (*e.g.*, adjust tail pointer, increment count of elements in queue etc.)

- Returns a SUCCESS code to be stored in the status register at the host (`status`)

The above sequence of steps may either be implemented entirely in hardware or in software/firmware with hardware providing only the support necessary to ensure atomicity.

An alternative implementation is to issue the enqueue operation as a simple store operation with `P` as the address and `D` as the data. The advantage of this approach is that the PIM capabilities can be used by processors that may not implement PIM-oriented instructions. In this implementation, the near-memory processor must check each incoming store operation's addresses against all registered queues in the module to detect enqueue operations. To reduce the overhead of doing so, a region of memory within the module may be set aside for queue metadata (the queue registration system software must abide by the same convention). Therefore, only addresses that fall with that region need to be checked against handles of queues.

In either of the above implementations, only accesses to the same queue need to be serialized (*i.e.*, the above sequence of operations need only be atomic with respect to other operations on the same queue). Other memory operations as well as operations to other queues may proceed in parallel.

Serialization among operations to the same queue is enforced at a point where all accesses to a given address (*e.g.*, `P` in this case) must go through (*e.g.*, the DRAM controller). Once a queue operation starts, all other operations to that queue are stalled at the serialization point until the in-flight operation completes. Some amount of buffering may be provisioned in the near-memory processor to hold stalled queue operations. If such buffering fills up, back-pressure may be applied to requestors. Alternatively, credit-based schemes may be used to ensure that the number of in-flight queue operations can be accommodated in the available buffering. Note that separate

credit pools may be used for queue operations and normal memory operations as normal memory operations are not stalled due to queue operations.

To dequeue a word from the queue, the application performs the following operation:

```
status = atomic_dequeue(P, D)
```

Here, `P` is the opaque queue handle returned when the queue was registered and `D` is the register to store the data payload word to be dequeued. If the operation succeeds, a SUCCESS code is stored in `status` upon completion. If the operation fails (*e.g.*, the queue is empty), an error code is stored in `status`.

The preferred implementation is for atomic_dequeue() to be implemented as a single instruction on the host processor. An example format for such an instruction is as follows:

```
adq rs, rd1, rd2
```

Here, the `adq` mnemonic identifies the atomic dequeue instruction and `rs` identifies the source operand: a register containing the queue handle (`P` in the above example). Note that alternate implementations may allow the source operand to be specified as constants encoded in the instruction or to be computed using registers and/or constants (*e.g.*, the queue handle formed by adding a constant operand from the instruction to a value in a register). Registers `rd1` and `rd2` specify where to place the two resulting values: the data value dequeued (`D`, in the above example) and the status of the operation.

When executed, the `adq` instruction generates a “dequeue” operation to the memory module to which the address pointed to by handle `P` maps to. `P` is passed as the input operand to the dequeue memory operation (much like the address operand of a load operation). Upon receiving

the dequeue(`P`) operation, the near-memory processor performs the following operations atomically with respect to other queue operations using the same handle `P`:

- Uses the operand (`P`, in this case) to access the queue metadata structure.
- If the metadata indicates the queue is empty:
    - Does not change anything in the queue and returns an error code to be stored in the status register at the host (`status`)
- Otherwise (*i.e.*, if the queue is not empty):
    - Read the next available data word from the data payload storage of the queue and return to be stored in the result register (`D`, in this case)
    - Update queue metadata as appropriate (*e.g.*, adjust head pointer, decrement count of elements in queue etc.)
    - Returns a `SUCCESS` code to be stored in the status register at the host (`status`)

The above sequence of steps may either be implemented entirely in hardware or in software/firmware with hardware providing only the support necessary to ensure atomicity.

Similar to atomic_enqueue(), atomic_dequeue() can be implemented as a separate instruction or via a load operation to the address `P`. In the latter case, also similar to enqueue, the PIM may then compare the address of known queues with the queue handle being passed in (`P`) to determine which loads correspond to queue operations.

Once a queue operation starts, all other operations to that queue are stalled at the serialization point (*i.e.*, DRAM controller) until the currently executing operation completes. Queue accesses are only serialized with respect to other accesses to the same queue. Normal load/store memory operations as well as operations to other queues can proceed in parallel.

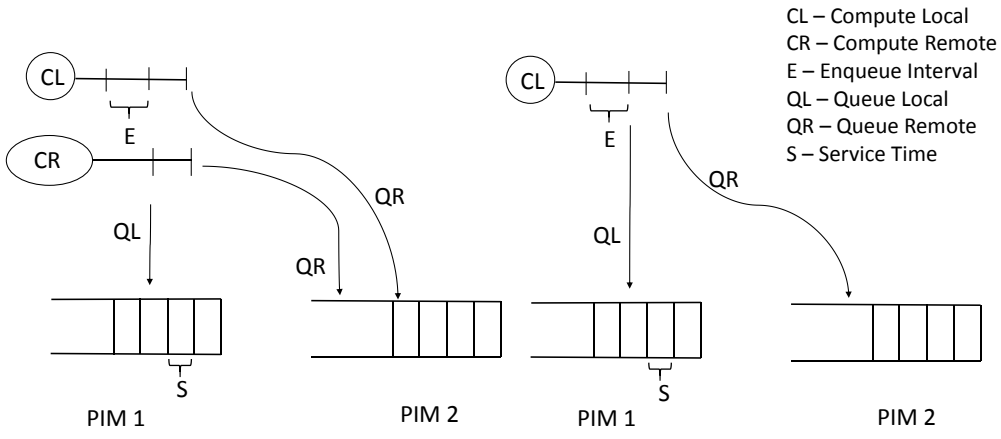
7.3 Evaluation Methodology

In this section I present the methodology for evaluating work migration against the *baseline*, where data is accessed remotely when needed and no work migration is done. First, I implement a queuing framework to support a distributed data-driven implementation of breadth-first search (BFS), single source shortest path (SSSP) and betweeness centrality (BC) in the system of study. Second, I look at possible ways to improve the queue efficiency by taking advantage of PIM. Third, I develop a timing model to compare work migration against the *baseline*. Using both the queuing framework and the timing model, I compare the performance of work migration vs. the *baseline*.

The main purpose of using this framework instead of a cycle-level simulator is to easily and quickly perform a rapid exploration of a large parameter space consisting of various hardware systems, algorithm characteristics, and data sets.

7.4 Timing Model

To compare the behavior of work migration vs. the *baseline*, I develop the high level timing



(a) (b)

Figure 22. (a) and (b) represent the baseline and the work migration scenario, respectively.

model shown in Figure 22. Figure 22(a) shows the *baseline*, where both the graph and the computation are partitioned among all the memory modules but the neighbors of a vertex that is being processed are enqueued in a round robin fashion to the queues of the various PIM devices; some vertices are processed locally and others are processed remotely (performing remote memory accesses to the vertex data that is located on a remote PIM stack). Figure 22(b) represents the work migration scenario, where both the graph and the computation are also partitioned among all the PIM stacks, but vertices are processed locally; some queue operations are performed to local queues and others to remote queues. When a vertex is processed their neighbors are enqueued to the PIM queue where their data is located, which may be the local queue or a queue located on a remote memory module.

The parameters shown in Figure 22 represent the number of cycles for each of the key operations in our system . *CL* and *CR* model computation time while the remaining parameters model various aspects of queueing and memory accesses. I use this parameterized model to study different application characteristics and a variety of hardware systems.

CL denotes the number of cycles spent doing computation on a vertex that is local to the PIM. Varying this parameter allows us to model applications with different amounts of computation per vertex. BFS is an example of an algorithm that performs no computation per vertex and has a low *CL* value. *CR* is the number of cycles spent doing computation on a vertex that is remote to the PIM in the *baseline* model. *CR* is not an independent parameter in our model and is computed as $CR = CL + X \cdot C$, which models the remote computation as consisting of the same computation as a local vertex plus some number of remote memory accesses. The total remote memory access overhead during the vertex's computation is expressed as the product of the remote memory access communication overhead *C* and the number of serialized remote

memory accesses *X* (sets of independent memory accesses are counted as a single access in *X* as those can be issued in parallel to overlap their latencies). Increasing the value of *C* allows us to model systems in which it takes longer to perform a remote memory access[3], and varying *X* models applications that present a different ratio of computation to remote memory accesses.

Parameter *QL* is the number of cycles it takes for a vertex to arrive at a queue that is in the local memory of the PIM device performing the enqueue operation. This parameter is based on the latency of accessing local main memory. *QR* is the number of cycles that it takes a vertex to reach a remote queue; I define $QR = QL + C$. The service time of a queue is the time it takes to enqueue or dequeue a vertex and I represent it as *S*. This parameter models potential overheads of using the queues. A naïve queue implementation, such as one that requires acquiring a global lock will have a high value of *S* while more efficient queue implementations will have lower values. *E* is the interval between back-to-back enqueues of vertices issued by a single PIM device. A low value of *E* corresponds to more frequent issue of enqueue operations and models PIM processors with higher operating frequencies or higher degrees of parallel execution. Note that lowering the value of *E* also increases the degree of contention seen by the queues. Even though multiple processors can issue an enqueue operation to the same queue, these operations

[3]Although my framework can be extended to model different communication delays (*i.e.*, different values of *C*) for each pair of communicating PIM devices, my results here use a constant remote access latency for all PIM devices. I expect this to yield reasonable results on average for applications with irregular neighbor remote memory accesses, such as the graph algorithms considered here.

are serialized at the queue and only one element can be accepted by the queue at each time interval (as defined by the parameter *S*); this way I correctly account for contention at the queue.

I initially assume that atomicity in queue manipulations is achieved via an atomic modification of the queue pointers (*i.e.*, atomic increment or decrement of queue pointers). I model the cost of these atomic operations as equivalent to the cost of a load to main memory that is serialized with the queue access. Some atomic operations will be performed to the local memory and some others to a remote memory, depending on what PIM the queue that is being accessed is located. I assume that reaching to any remote PIM constitutes the same amount of communication overhead (*C*).

7.4.1 Experimental Setup

With the queuing framework and the timing model I first study the benefits of using PIM to serialize the access to the queues and later I compare the performance of work migration vs. the *baseline* for the selected graph traversal algorithms. I study the effect of load imbalance and compare the energy of the memory accesses for the different approaches. I study a set of algorithms that are core primitives for graph processing and are fundamental for many other more complex applications.

For my experiments I choose parameters within a range that is reasonable for the study system. I choose the latency to access local memory (*QL*) to be 64 cycles. I vary the latency of accessing remote memory (*QR*) from 64 to 640 cycles, sufficient to cover a broad range of design options from multiple memory modules within a single package to various interconnection network options among memory modules in separate packages. I choose

parameter $E = 1$, as I model processors that can enqueue one element per cycle. Parameter $S = 1$, as the queues can accept one element per cycle.

I measure the number of local and remote memory accesses for each approach and compare the total relative memory energy considering the relative energy of a remote memory access with respect to a local memory access. Similar to the performance modeling, I consider a range of remote memory access energies that span a broad range of possible implementations. I compare the total memory energy for the cases where the energy of a remote memory access is 1x, 5x and 10x that of a local memory access.

7.4.2 Graph Algorithms

Breadth First Search (BFS) is a fundamental graph traversal algorithm. It traverses all the vertices in a graph that accessible from the starting source vertex. Initially all vertices are set as not visited and the traversal begins at the source vertex. First, the neighbors of the source vertex are visited, which are at a distance 1 from the source. Then all the neighbors of the vertices at distance 1 that have not been visited before are visited. The same operations are repeated until all vertices reachable from the source vertex have been visited, always visiting all vertices at distance k from the source before visiting any vertex at distance k+1.

Single Source Shortest Path (SSSP) is a graph algorithm that computes the shortest distance from each vertex in the graph to the source vertex. I use a modification of the Bellman-Ford algorithm, which works for graphs that have positive edge weights [67]. Each vertex keeps track of the shortest distance to the source vertex, which is initially set to infinity. This value keeps being updated in an iterative way as new paths to a vertex are found that have shorter distance.

Table 7. Main Characteristics of the Graphs

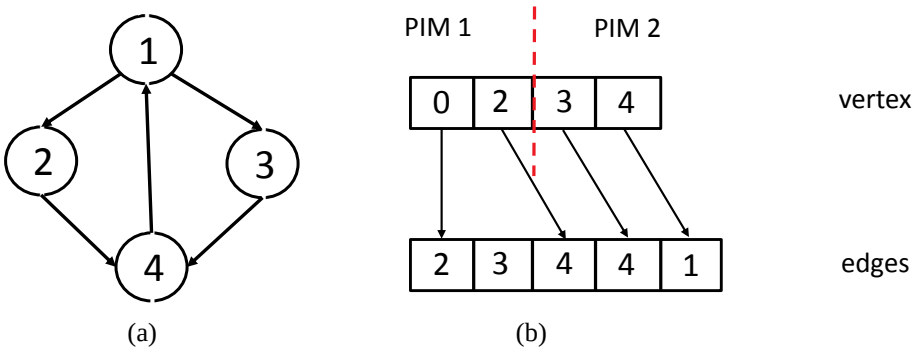
Name	Description	# Vertices	# Edges	Degree	Distribution	Diameter
Road	Roads of Texas	1,379,917	1,921,660	1.4	normal	1054
Amazon	Co-purchased products	403,394	3,387,388	8.4	normal	35
Pokec	Social network	1,632,803	30,622,564	18.7	power	14

Betweeness centrality (BC) is a graph algorithm used in social networks to compute the importance that each vertex has on the graph. The BC value of a vertex is directly proportional to the number of shortest paths that pass through that vertex. This graph algorithm requires computing as many BFS graph traversals as the number of vertices in the graph, each of the BFSs traversals starts at a different vertex. During the BFS traversal the number of shortest paths passing through each vertex is calculated. There are two phases in this algorithm; first, the number of shortest paths passing through each vertex is computed while doing a forward BFS traversal starting at each of the vertices, and second, a backtracking BFS traversal is done where the dependencies are computed.

My implementation uses the algorithm proposed by Madduri *et al*. [68]. Their algorithm is a modification of the traditional parallel algorithm proposed by Brandes [69] and it manages a lock-free implementation by tracking the successors of each vertex instead of the predecessors.

7.4.3 Compressed Sparse Row Representation

I use the compressed sparse row (CSR) representation to store the structure of the graphs. In this representation the vertices are stored in one array and the edges are stored in a different one; the vertex array stores the offset into the edge array, giving the position of the first outgoing edge of each vertex in the edge array. Figure 23 shows this representation. The main advantage of CSR is that it is compact and it can be easily divided uniformly among the different PIMs.

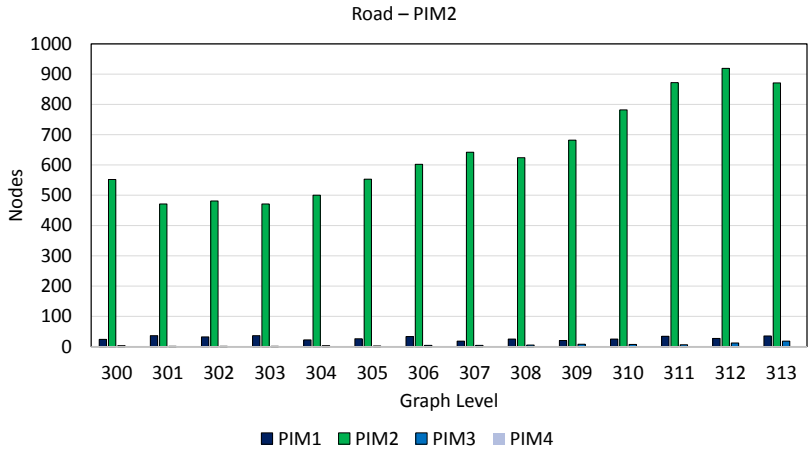


(a) (b)

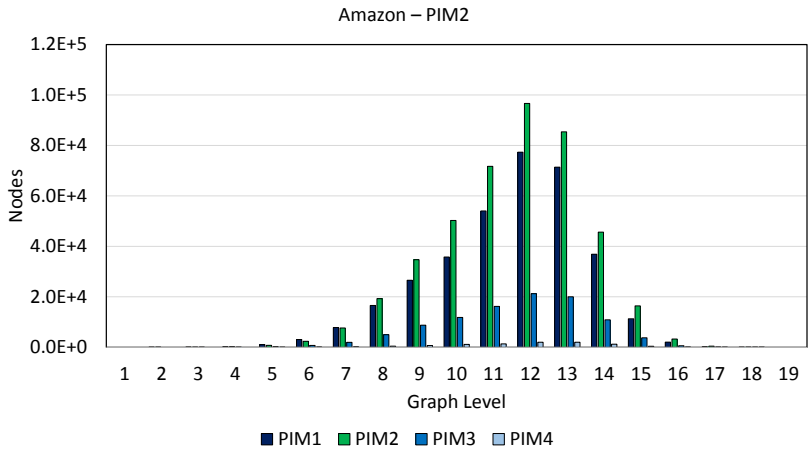
Figure 23. (a) the graph and (b) its CSR representation with the vertex and edges arrays.

7.4.4 Workloads

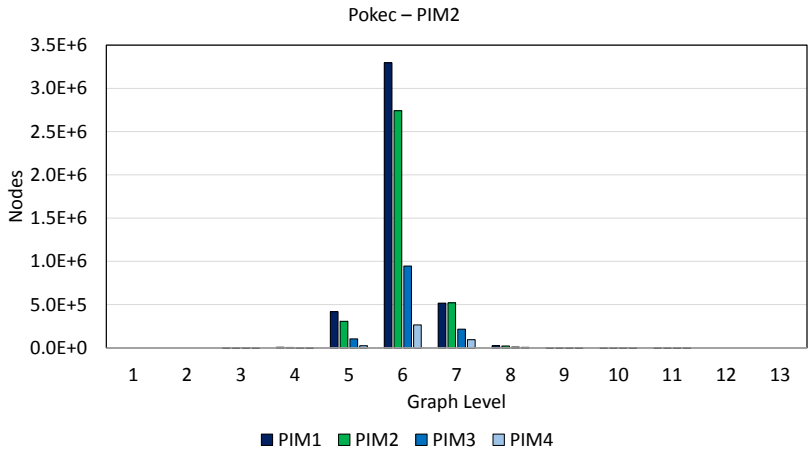
I use three real-world graphs from SNAP [66], described below, that present different characteristics. Table 7 summarizes their main characteristics. The diameter of a graph is the longest shortest path in the graph. There are two types of distributions: *normal*, in which most vertices have approximately the same number of vertices, and *power-law*, in which some vertices have a very large number of edges while most vertices have just a few.



(a)



(b)



(c)

Figure 24. Neighbor distribution when the graph that has been partitioned among four PIM stacks. Graph level is the distance from the root vertex. This figure shows the spatial distribution of the neighbors for the vertices at each level of the graph that are located on PIM2

Texas road network graph: This graph shows intersections of roads in the Texas road network as vertices and the roads between these intersections as edges.

Amazon co-purchasing graph: This graph shows combinations of products that are usually bought jointly on Amazon. The vertices are the different products in Amazon and the edges show if they were bought together.

Pokec social network graph: Pokec is a social network in Slovakia that consisting of more than 1.6 million users, making it the most used social network in the country. The vertices are the individuals in the social network and the edges show the relationships between them.

As Table 7 shows these graphs have different characteristics, and this is the reason they were chosen for the study. Figure 24 shows for the three graphs of study the neighbor distribution for the vertices that are located in PIM2, when the graph data structure has been partitioned evenly among 4PIMs. This data helps to understand the locality in neighboring vertices of the graph. Figure 24(a) shows the neighbor distribution for Texas road graph, I can see that most of the neighbors of the vertices in PIM2, are also in PIM2, only a very small number are in PIM1 and PIM3 and none in PIM3, this shows that Texas road graph has very high locality in the neighbors. Figure 24(b) shows the neighbor distribution for Amazon, I can see in this graph that most of the neighbors are located in PIM2, but also a considerable number of neighbors are in PIM1 and PIM3, so this graph has medium locality in the neighbors. Figure 24(c) shows the neighbor distribution of Pokec, I can see that most of the neighbors are located in PIM1, and also a large number in PIM3, this graph presents the lower locality of the three graphs.

7.5 Evaluation Results

In this section I compare the performance of work migration vs. the *baseline* case by applying the timing model to the graph traversal algorithms. I first study the performance impact of using PIM to serialize the queue operations without the need for application-level atomics. Later, I study the performance of work migration vs. the *baseline* and I propose modifications to improve its performance. Finally, I study load imbalance in our graph traversal algorithms and mechanisms to improve load balancing while maintaining locality.

In this section I study the effect of my proposed mechanisms to take advantage of the proximity of PIM to the memory to serialize the queue operations guaranteeing atomicity without the use of explicit software atomics.

7.5.1 Hardware Support for Efficient Shared Queues

Figure 25 shows the performance difference of using explicit software atomics (sw_atomics) to serialize the queue operations vs. using PIM (pim_queues). I present results for BFS for the Amazon and Pokec graphs, when doing migration for 4, 8 and 16 PIMs. For these experiments I vary the cost of accessing remote memory *QR* from 64 to 640 cycles, which is 1x to 10x the cost of accessing local memory *QL*. The results are normalized to the performance of pim_queue for 4 PIMs. I observe that for all cases my proposed mechanism performs better, with the Amazon graph demonstrating a larger relative benefit, up to 16.7% improvement for 4PIMs when

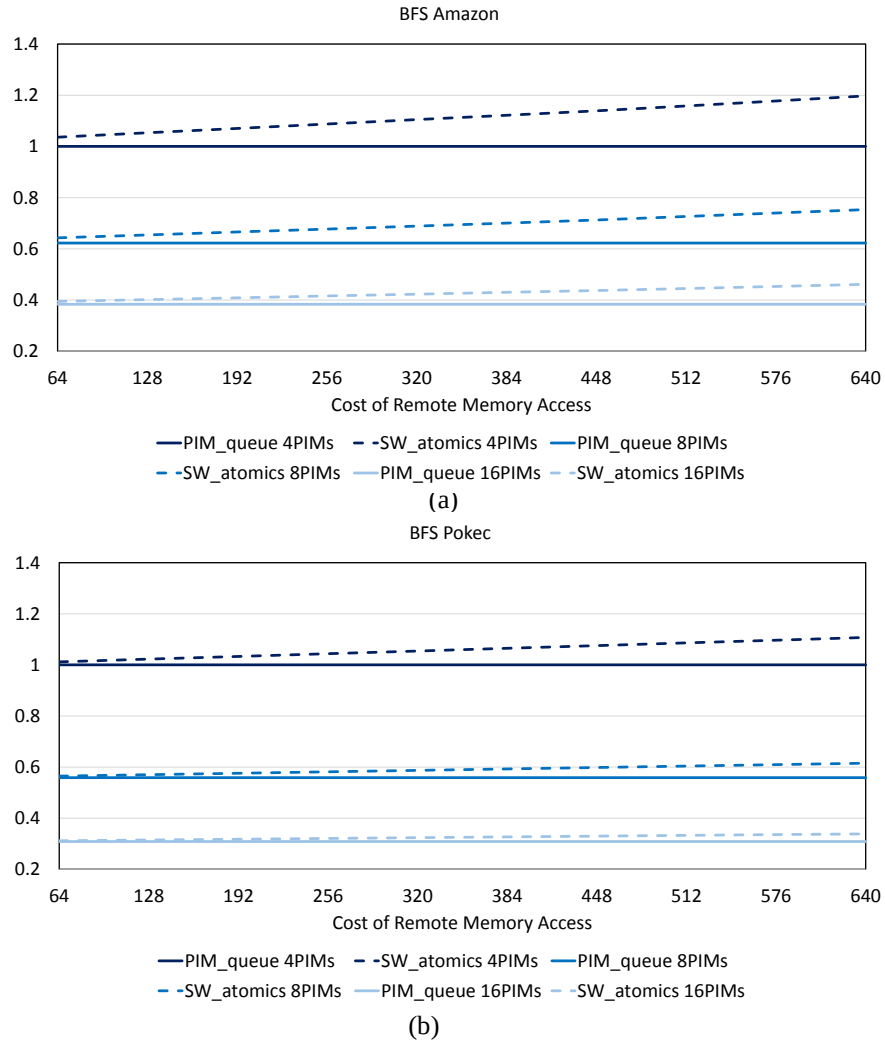


Figure 25. Performance of using explicit software atomics to update the queues vs. using my proposed mechanisms that uses PIM to serialize the queue operations. I vary the latency to a remote PIM from 64 to 640 cycles (QR), which is 1x to 10x the latency to a local PIM (QL). Results are normalized to 4PIMs.

QR=640, due to the graph having a larger diameter. The maximum improvement that I see for Pokec is 9.7% for 4PIMs when *QR*=640. A clear advantage is that when avoiding the use of explicit software atomics, the performance of BFS is not dependent on the latency to access remote memory in a remote PIM. This is because I can eliminate the remote memory accesses from the critical path, as will be explained in the next section. The rest of the results I use of this hardware support.

7.5.2 Study of Work Migration

In this section I study the performance of work migration compared to the *baseline*. I also compare the number of local and remote memory accesses performed by each of the policies.

I consider two different alternatives to implement BFS and SSSP. When processing a vertex these algorithms only require data related to that vertex or data from the immediate predecessor vertex (SSSP requires the distance of the predecessor vertex to the root in order to compute the distance of the current vertex to the root). First, I avoid the need to perform remote memory accesses in the critical path by enqueuing all the neighbors of the vertex that is being processed to the corresponding PIM. Checking whether those vertices need to be processed or not is done in the next iteration of the algorithm locally. For SSSP the distance to the predecessor can also be enqueued with the vertex, I do this by merging both words together into a longer word that can be enqueued and dequeued atomically. The second alternative first checks whether the neighboring vertices of the vertex that is currently being processed need to be processed or not before enqueuing them. This ensures that only the vertices that need to be processed are enqueued, but requires performing remote memory accesses in the cases where the neighbors are located in different PIM stacks. I call these two implementations *migration* and *migration remote*:

Migration: This policy eliminates remote memory accesses from the critical path by enqueuing all the neighbors of the vertex that is being processed to their corresponding PIM. The check is later done locally. This policy results in vertices being enqueued and dequeued that might not need to be processed, but avoids remote memory accesses in the critical path.

Migration remote: This policy performs remote memory accesses to check if the neighboring vertices need to be processed. If a vertex needs to be processed it is enqueued to the corresponding PIM. This policy avoids enqueuing and dequeuing extra vertices at the cost of having remote memory accesses in the critical path.

In the case of BC, when a vertex is being processed data from its successors is needed. So memory accesses to the successors are unavoidable and these memory accesses can be local or remote. For BC I use the second implementation alternative.

Enqueueing and dequeueing all the neighboring vertices to later check whether they need to be processed can be expensive. The higher degree a graph has, the more expensive *migration* becomes. I propose the following modifications to *migration* in order to reduce the number of vertices that are enqueued and still avoid remote memory accesses in the critical path:

Migration local: Similar to *migration*, but this policy checks the vertices that are local to the PIM to ensure they need to be processed in the next iteration before enqueuing them. Since the vertex is local, the memory accesses to the vertex's data are local. Vertices that are not local are still enqueued without validation.

Migration duplicates: This policy is similar to *migration*, but the queues are implemented as a hash table to eliminate duplicate vertices. Every time an element is enqueued, the hash table is looked up to see if the element is already present, and if it is not the element would be written

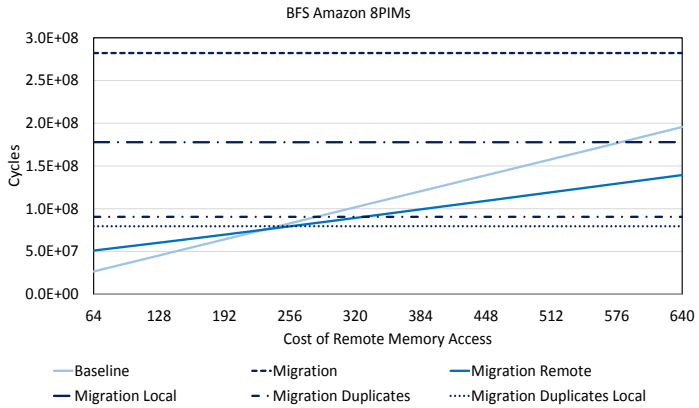
to the table. This results in extra memory accesses when enqueuing, but they are overlapped with the computation. This table could be implemented in software. Accesses to the hash table would still be serialized by my previously proposed hardware mechanisms introduced in Section 7.2.3.

Migration duplicates local: Combination of *migration local* and *migration duplicates*.

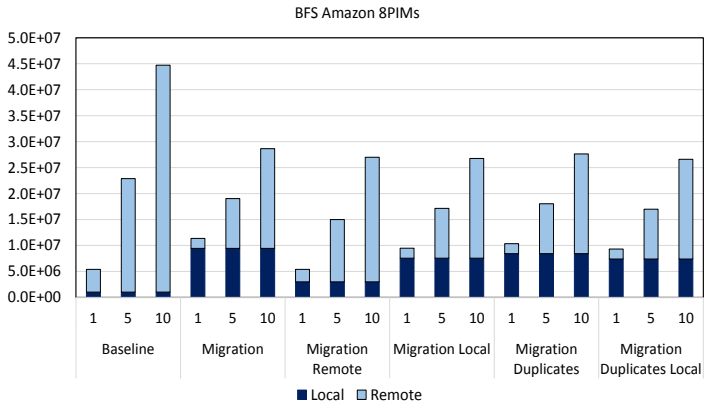
Figure 26 shows the performance and memory accesses of the different migration policies vs. the *baseline* for BFS, SSSP and BC, as I increase the cost of accessing remote memory. I vary the cost of accessing remote memory (*QR*) from 64 to 640 cycles. I compute the total memory access energy as the energy of a remote memory access is 1x, 5x and 10x that of a local memory access. I study the behavior of the Amazon, Texas road and Pokec graphs, when using 4 and 8 PIM devices. I only show a subset of the results due to space constraints. The presented data is sufficiently representative of the rest and can help understand the tradeoffs of the different policies. For BFS and SSSP, Figure 26 (a) and (c), I can use the *migration* policy where all the neighboring vertices of the vertex being processed are enqueued, and checking whether the vertex has been visited (or whether the distance to the vertex is lower, for SSSP) is done locally. *Migration* results in a large number of local memory accesses and only some enqueue operations are remote. As the remote queue operations are overlapped with the computation, the performance of *migration* is not affected by the increasing cost of accessing remote memory. But the performance of *migration* is worse than that of the *baseline* and *migration remote*. Although the number of remote memory accesses is low for *migration*, the large number of local memory accesses results in high total memory energy, as Figure 26(b) and (d) show.

In Figure 26(a) and (c) I observe that the performance of *migration* improves with the proposed modifications (*migration local*, *migration duplicates* and *migration duplicates local*)

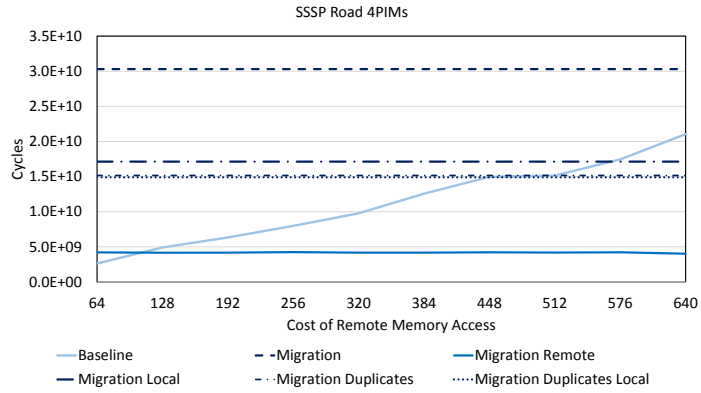
and they are also not affected by the cost of accessing remote memory. On the contrary, the performance of *baseline* worsens as the cost of accessing remote memory increases. At lower *QR* the *baseline* performs better, as it achieves better load balance among PIMs by enqueuing vertices in a round robin way, and the effect of *QR* is lower and less important than the impact of load imbalance. The *baseline* performs multiple remote memory accesses when processing a vertex remotely, to read the graph data structure that is stored in remote PIMs. These accesses are often interdependent resulting in multiple serialized round trip accesses to the remote PIM. For example, the vertex array needs to be accessed first to find the corresponding index to the edge array, which contains the neighbors of a vertex. On the contrary, *migration* and its variants are not sensitive to higher latency to remote PIM stacks as accesses to a remote PIM are performed when a vertex is enqueued to a remote queue, which is off the critical path. These new policies based on *migration* result in fewer number of local memory accesses than *migration* as shown in Figure 26 (b) and (d), lowering the total memory energy.



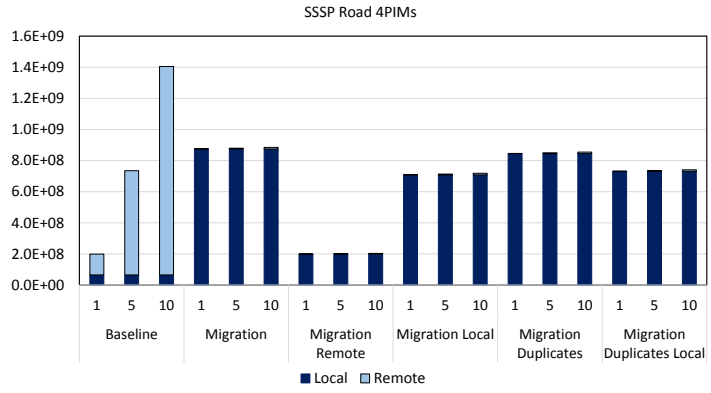
(a)



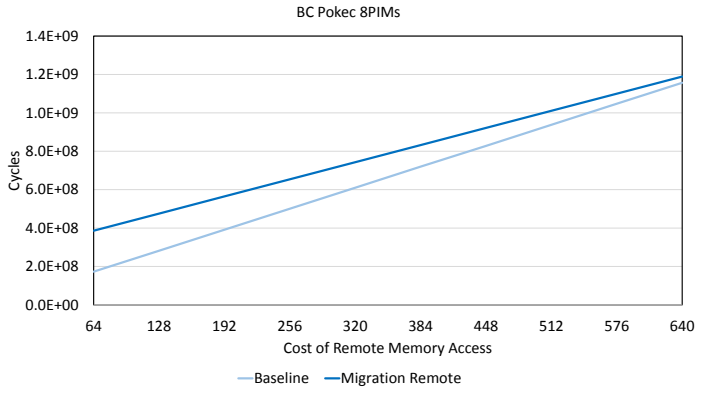
(b)



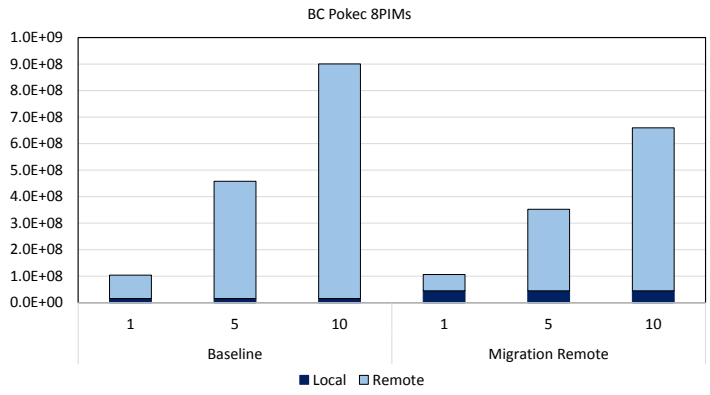
(c)



(d)



(e)



(f)

Figure 26. The first column shows the performance of BFS for the Amazon graph (8PIMs), SSSP with Texas road network (4PIMs) and BC for the Pokec social network (8PIMs). I vary the latency to a remote PIM from 64 to 640 cycles (*QR*), which is 1x to 10x the latency to a local PIM (*QL*). The second column shows the energy of the memory accesses when the energy of a remote memory access is 1x, 5x and 10x that of a local memory access.

In Figure 26(a) I observe that the performance of *migration remote* worsens with *QR*. This is due to the remote memory accesses that this policy performs to check what vertices to enqueue. For low *QR*, *baseline* performs better than *migration remote* as it is more balanced, but as *QR* increases a low number of remote memory accesses becomes more important. The number of remote memory accesses for *migration remote* is however larger than for the other migration policies; as can be seen in Figure 26(b). This is not the case for SSSP with the Texas road graph; as Figure 24(c) shows. The Texas road network graph presents few edges per vertex, and high locality in its neighbors, meaning that the neighbors of a vertex are likely to be on the same PIM stack as the parent. Therefore, *migration remote* is not affected by *QR* as most of the memory accesses are local. This is why *migration remote* results in better performance than any of the other migration policies. As *migration remote* does check and, if necessary, updates the distance to the neighboring vertices before enqueuing the vertex, only the vertices that need to be processed are enqueued, considerably reducing the number of memory accesses.

In Figure 26(e) for BC, I use *migration remote*; I need to perform memory access and update a vertex's data before it is enqueued. This is the reason why *migration remote*'s performance increases with the cost of accessing remote memory. In this case *migration remote*'s performance is worse than the performance of the *baseline* due to the load imbalance. The degree of imbalance is input-dependent, but I observe it in all the studied graphs. As the number of PIMs increases, load imbalance goes up. Figure 26(f) shows the memory accesses performed by both policies. I observe that the number of remote memory accesses is larger for the *baseline* than for *migration remote*, the *baseline* results in more energy spent in memory accesses. In the next section I explore the effect of work stealing to improve the performance of *migration remote* while still trying to maintain locality.

7.5.3 Load Balancing and Locality

As shown before for the BC case, the *baseline* performs better than *migration remote* although the *baseline* results in more remote memory accesses. For BFS and SSSP, the *baseline* performs better than the rest of the policies for low *QR*. For all cases this is due to the load imbalance in the graph algorithms and the graphs.

In this section I study *work stealing* implemented on top of *migration remote* and *DL work stealing* implemented on top of *migration duplicates local* to address the load imbalance problem. I implement a traditional work stealing mechanism, where threads first execute the vertices that are in their local queue and once their local queue is empty, if there is still work to be completed in the system, they steal from the queue with the most elements. When the cost of accessing remote memory and local memory is close, work stealing reduces load imbalance and the number of vertices processed by each PIM is similar. As the cost of accessing remote memory increases, the cost of processing vertices remotely goes up, and fewer vertices will be processed remotely, naturally reducing the degree of work stealing and decreasing the number of remote memory accesses. Work stealing becomes more efficient as the cost of accessing remote memory goes up.

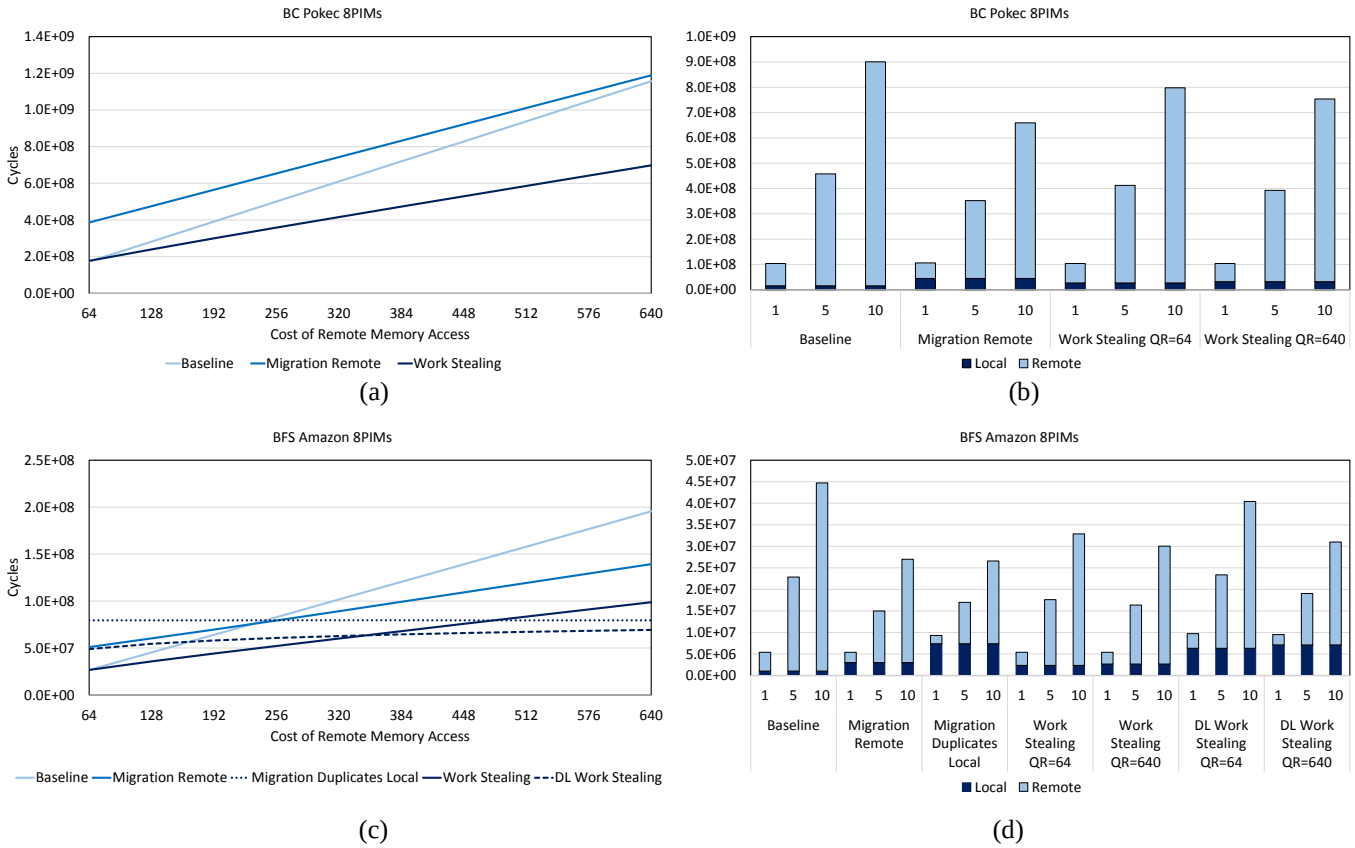


Figure 27. The first column shows the performance of work stealing for BC with the Pokec graph (8PIMs) and work stealing and DL work stealing for BFS with Amazon graph (8PIMs). I vary the latency to a remote PIM from 64 to 640 cycles (QR), which is 1x to 10x the latency to a local PIM (QL). The second column shows the energy of the memory accesses when the energy of a remote memory access is 1x, 5x and 10x the energy of a local memory access.

Figure 27 shows the performance of *work stealing* for BC using the Pokec graph and both *work stealing* and *DL work stealing* for BFS using the Amazon graph. Figure 27(a) shows the results for BC, and *work stealing* always performs better than the *baseline*. This is because *work stealing* first executes as many vertices locally as possible and then it executes vertices remotely (stealing vertices from other queues) to balance the load of the different threads. Figure 27(c) shows the results for BFS, *work stealing* is the best performing policy for lower *QR*. For higher *QR*, *DL work stealing* performs better as it is not affected by *QR*. Figure 27(b) and (d) show the memory accesses performed by the different policies and the total memory access energy as the energy of a remote memory access is 1x, 5x and 10x that of a local memory access. I see that *work stealing* performs more remote memory accesses and less local memory accesses than

migration remote, but less remote memory accesses than the *baseline*. In Figure 27(b) I see that *DL work stealing* presents more local memory accesses than *work stealing*, but less than *migration duplicates local*. Figure 27(b) and (d) show that for both work migration policies the number of remote memory accesses decreases as the cost of accessing remote memory goes up resulting in lower total memory energy with *QR*.

I observe that there is a tradeoff between performance and energy; both *work stealing* and *DL work stealing* provide better performance for higher energy than other policies such as *migration remote or migration duplicates local*, since in order to balance the load it needs to steal vertices and execute them remotely, but this difference decreases as *QR* increases.

7.6 Summary

This work shows that the large amount of irregular memory accesses present in graph algorithms make it challenging to implement these algorithms obtaining high data locality on systems with multiple PIM devices. I propose fine-grained work migration to maximize data locality in PIM-based systems.

This work proposes a framework to explore a large space of graph application behaviors and system architectures. I provide insight regarding what characteristics of the graph applications and system parameters result in efficient work migration to take advantage of data locality.

Hardware support for efficient queue implementations can considerably increase the performance of my work migration mechanisms. I present a technique for such hardware support that leverages the DRAM controller in PIM to enforce serialization of operations to the same queue, eliminating the overheads of atomics. My method also does not require dedicated

hardware storage, allowing it to scale to arbitrary numbers of queues and arbitrarily large queues, limited only by the DRAM capacity of the memory modules.

To conclude I can say that fine-grain task migration results in lower number of remote memory accesses in graph algorithms compared to the baseline, but naïve migration schemes increase the number of local memory accesses which can hurt both performance and energy. I proposed optimizations to fine-grain task migration to reduce the number of vertices that are enqueued/dequeued, mitigating the increase in local memory accesses and still keeping remote memory accesses of the application'scritical path. I also explored a hybrid scheme with some amount of remote memory accesses and task migration (*Migration remote*), which also reduces the number of local memory acceses but its performance worsens with remote memory access latency. I also evalauted work stealing mechanisms to further improve the performance of fine-grain task migration by reducing the load imbalance at the cost of some loss of data locality in the memory accesses.

8 Conclusions

In this thesis I studied different task allocations and migration techniques in novel architectures, and their implications on performance, energy savings and QoS and fairness. With the slowdown of Moore's Law and Dennard scaling, power consumption is currently the most important design constraint for increasing microprocessors' performance. New architectures that result in higher power efficiency are being researched to continue increasing computing performance. In this thesis I focused on two new architectural trends: GPGPU computing and PIM systems. I showed that these architectures present new challenges when it comes to scheduling work on them and I demonstrated how smart scheduling decisions considerably increase the performance of these novel architectures.

For GPGPU execution, I showed that spatial multitasking improves overall system performance and resource utilization compared to cooperative multitasking. In this thesis I studied how to distribute computing resources to different concurrently-executing GPGPU applications considering process variation, QoS and fairness.

I first proposed a variation-aware assignment of SMs to applications in a GPU that supports spatial multitasking and PSMC. I characterized GPGPU applications based on their sensitivity to frequency and applied a variation-aware assignment of SMs to applications. The proposed technique that assigns faster SMs to compute-bound applications and slower SMs to memory-bound applications demonstrated an improvement in performance. I also evaluated different SM partitioning algorithms that take advantage of PSMC in a multitasking GPU. When executing two or three applications concurrently, I achieved speedups of 18%-20% using *Even-Split* and *Profile* partitioning algorithms, respectively, compared to the same partitioning algorithms with

global clocking. These speedups increased to 36%-46% when compared to the performance of cooperative multitasking with global clocking.

Spatial multitasking on the GPU can also help satisfy the performance requirements of QoS applications by assigning them enough computing resources to meet those requirements. The remaining SMs can then either be allocated to executing best-effort applications to improve overall system performance, or be left idle to save power. I showed that this allocation is complicated by the fact that the performance of the QoS application depends on the co-executing applications, and the consequent contention on the shared resources. This is the reason why a static allocation of resources to the applications is not a good approach. I showed how a linear approximation algorithm allows us to dynamically allocate the minimum number of resources to the QoS application, so that it satisfies its performance requirements no matter what other applications run simultaneously on the GPU. By using spatial multitasking with QoS applications on the GPU I obtained 7W power consumption reduction or 17.57% performance improvement for co-executing best-effort applications.

Next, I studied fairness in the resource allocation in GPGPUs. I showed that maximizing system performance can come at a significant cost of fairness of individual application performance. This work introduced a fairness metric that partitions the SMs between simultaneously-executing applications to increase performance relative to cooperative multitasking, and do so without sacrificing individual application performance. I further presented a methodology to determine at runtime how to allocate GPU resources to meet this metric, and adjust that allocation if isolation-based performance predictions are determined to be inaccurate in the face of multiple co-executing applications.

Last, I performed a high-level analysis of graph algorithms executing in a system that consists of multiple memory devices with PIM capabilities. The irregular memory accesses patterns of these applications can result in large number of remote memory accesses, negating the benefits of PIM. I proposed fine-grained work migration to maximize data locality in PIM-based systems. I also proposed hardware support for efficient queue implementations taking advantage of the proximity of PIM to the memory.

I found that fine-grain task migration results in lower number of remote memory accesses in graph algorithms compared to the baseline, but naïve migration schemes increase the number of local memory accesses which can hurt both performance and energy. I proposed optimizations to fine-grain task migration to reduce the number of vertices that are enqueued/dequeued, mitigating the increase in local memory accesses and still keeping remote memory accesses of the application'scritical path. I also evaluated work stealing mechanisms to further improve the performance of fine-grain task migration by reducing the load imbalance at the cost of some loss of data locality in the memory accesses.

As these new architectures become adopted in more computing systems, and a large variety of applications are modified to execute on them, it will become increasingly important for these architectures to provide better support for resource allocation, task partitioning and task migration will increase in importance. Properly implementing these capabilities will determine the performance and energy benefits that these architectures will be able to provide—and in some degree will further help with their adoption.

9 Publications

P. Aguilera, D. P. Zhang, N. S. Kim and N. Jayasena, “Fine-Grained Task Migration for Graph Algorithms using Processing in Memory”, *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, Chicago, IL, USA, 2016, pp. 489-498.

P. Aguilera, N. S. Kim and K. Morrow, “Fair Share: Allocation of GPU Resources for Both Performance and Fairness,” *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, Seoul, 2014, pp. 440-447.

P. Aguilera, J. Lee, A. Farmahini-Farahani, K. Morrow, M. Schulte and N. S. Kim, "Process Variation-aware Workload Partitioning Algorithms for GPUs Supporting Spatial Multitasking", *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Dresden, 2014, pp. 1-6.

P. Aguilera, K. Morrow and N. S. Kim “QoS-Aware Dynamic Resource Allocation for Spatial-Multitasking GPUs”, *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Singapore, 2014, pp. 726-731.

D. Chang, C. Jenkins, P. Garcia, S. Gilani, **P. Aguilera**, A. Nagarajan, M. Anderson, M. Kenny, S. Bauer, M. Schulte, K. Compton “ERCBench: An Open-Source Benchmark Suite for Embedded and Reconfigurable Computing”, *2010 International Conference on Field Programmable Logic and Applications*, Milano, 2010, pp. 408-413.

References

[1] Khronos Group, 2011. [Online]. Available: http://www.khronos.org/opencl/.

[2] "NVIDIA CUDA C Programming Guide," 2014. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[3] J. T. Adriaens, K. Compton, N. S. Kim and M. J. Schulte, "The case for GPGPU spatial multitasking," in *HPCA*, 2012.

[4] S. Borkar and A. A Chien, "The Future of Microprocessors," in *Communications of the ACM*, 2011.

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

[6] S. Keckler, W. Dally, B. Khailany, M. Garland and D. Glasco, "GPUs and the Future of Parallel Computing," in *MICRO*, 2011.

[7] W. J. Dally and J. Nickolls, "The GPU Computing Era," *IEEE Micro*, 2010.

[8] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler and T. W. Keller, "Energy management for commercial servers," in *Computer*, 2003.

[9] W. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *ACM SIGARCH*, 1995.

[10] A. N. Udipi, N. Muralimanohar, N. Chatterjee, R. Balasubramonian, A. Davis and N. P. Jouppi, "Rethinking DRAM design and organization for energy-constrained multi-cores," in *ISCA*, 2010.

[11] J. Carter and e. al, "Impulse: Building a smarter memory controller," in *HPCA*, 1999.

[12] J. Draper and e. al, "The architecture of the DIVA processing-in-memory chip," in *ICS*, 2002.

[13] S. Borkar, T. Karnik and V. De, "Design and reliability challenges in nanometer technologies," in *DAC*, 2004.

[14] J. Lee, P. Ajgaonkar and N. S. Kim, "Analyzing throughput of GPGPUs exploiting within-die core-to-core frequency variation," in *ISPASS*, 2011.

[15] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," 2009.

[16] NVIDIA, "NVIDIA’s Next Generation CUDA Compute Architecture: Kepler GK110, The Fastest, Most Efficient HPC Architecture Ever Built," 2012.

[17] NVIDIA, "NVIDIA GeForce GTX 980: Featuring Maxwell, The Most Advanced GPU Ever Made," 2014.

[18] P. Rogers, "The Programmer’s Guide to the APU Galaxy," 2011.

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

[20] S. Herbert and D. Marculescu, "Characterizing Chip-Multiprocessor Variability-Tolerance," in *DAC*, 2008.

[21] N. Kurd, J. Douglas, P. Mosalikanti and R. Kumar, "Next generation Intel® micro-architecture (Nehalem) clocking architecture," in *IEEE Symp. on VLSI Circuits*, June 2008.

[22] K. Bowman, A. Alameldeen, S. Srinivasan and C. Wilkerson, "Impact of die-to-die and

within-die parameter variations on the throughput distribution of multi-core processors," in *ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, 2007.

[23] E. Humenay, D. Tarjan and K. Skadron, "Impact of Process Variations on Multicore Performance Symmetry," in *DATE*, 2007.

[24] K. Jeffay, D. Stone and F. Smith, "Kernel support for live digital audio and video," in *Computer Communications*, 1992.

[25] D. Clark, S. Shenker and L. Zhang, "Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanism," in *SIGCOMM*, 1992.

[26] F. Harada, T. Ushio and Y. Nakamoto, "Adaptive Resource Allocation Control for Fair QoS Management," in *IEEE Transactions on Computers*, 2007.

[27] C. Poellabauer, L. Singleton and K. Schwan, "Feedback-based dynamic voltage and frequency scaling for memory-bound real-time applications," in *RTAS*, 2005.

[28] S. Hong and H. Kim, "An integrated GPU power and performance model," in *ISCA*, 2010.

[29] S. Kato, K. Lakshmanan, R. Rajkumar and Y. Ishikawa, "TimeGraph: GPU scheduling for real-time multi-tasking environments," in *USENIXATC*, 2011.

[30] M. Bautin, A. Dwarakinath and T.-c. Chiueh, "Graphic Engine Resource Management," in *MMCN*, 2008.

[31] A. Jog, E. Bolotin, Z. Guz, M. Parker, S. W. Keckler, M. T. Kandemir and C. R. Das, "Application-aware Memory System for Fair and Efficient Execution of Concurrent GPGPU Applications," in *(GPGPU-7)*, 2014.

[32] M. Bhadauria and S. A. McKee, "An Approach to Resource-aware Co-scheduling for CMPs," in *ICS*, 2010.

[33] K. Nesbit, N. Aggarwal, J. Laudon and J. Smith, "Fair Queuing Memory Systems," in *MICRO*, 2006.

[34] S. Kim, D. Chandra and Y. Solihin, "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture," in *PACT*, 2004.

[35] F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez and M. Valero, "Architectural Support for Real-time Task Scheduling in SMT Processors," in *CASES*, 2005.

[36] E. Ebrahimi, C. J. Lee, O. Mutlu and Y. N. Patt, "Fairness via Source Throttling: A Configurable and High-performance Fairness Substrate for Multi-core Memory Systems," in *ASPLOS*, 2010.

[37] D. A. Patterson, "Latency Lags Bandwith," *Commun. ACM*, 2004.

[38] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu and M. Ignatowski, "TOP-PIM: Throughput-oriented Programmable Processing in Memory," in *HPDC*, 2014.

[39] G. H. Loh, "3D-Stacked Memory Architectures for Multi-core Processors," *ISCA*, 2008.

[40] P. Jacob and a. et, "Mitigating Memory Wall Effects in High-Clock-Rate and Multicore CMOS 3-D Processor Memory Stacks," *Proceedings of the IEEE*, 2009.

[41] D. Chang, B. Gyungsu, K. Hoyoung, A. Minwook, R. Soojung, N. Kim and M. Schulte, "Reevaluating the latency claims of 3D stacked memories," in *ASP-DAC*, 2013.

[42] J. T. Pawlowski, "Hybrid memory cube (HMC)," *Hot Chips*, 2011.

[43] "Hybrid Memory Cube Specification 1.0," 2013.

[44] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *ACM SIGMOD International Conference on Management of data (SIGMOD '10)*, 2010.

[45] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola and J. M. Hellerstein, "Distributed GraphLab: a framework for machine learning and data mining in the cloud," in *VLDB Endow*, 2012.

[46] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams and P. Kalnis, "Mizan: a system for dynamic load balancing in large-scale graph processing," in *ACM European Conference on Computer Systems (EuroSys '13)*, 2013.

[47] S. Salihoglu and J. Widom, "GPS: a graph processing system," in *International Conference on Scientific and Statistical Database Management (SSDBM)*, 2013.

[48] J. Ahn, S. Hong, S. Yoo, O. Mutlu and K. Choi, "A Scalable Processing-in-memory Accelerator for Parallel Graph Processing," in *ISCA*, 2015.

[49] R. Nasre, M. Burtscher and K. Pingali, "Data-Driven Versus Topology-driven Irregular Computations on GPUs," in *IPDPS*, 2013.

[50] T. von Eicken, D. E. Culler, S. Copen Goldstein and K. Erik Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation," in *ISCA*, 1992.

[51] V. Agarwal, F. Petrini, D. Pasetto and D. A. Bader, "Scalable Graph Exploration on Multicore Processors," in *SC*, 2010.

[52] J. Chhugani, N. Satish, C. Kim, J. Sewall and P. Dubey, "Fast and Efficient Graph Traversal Algorithm for CPUs: Maximizing Single-Node Efficiency," in *IPDPS*, 2012.

[53] A. Buluc and K. Madduri, "Parallel Breadth-first Search on Distributed Memory Systems," in *SC*, 2011.

[54] J. Y. Kim and C. Batten, "Accelerating Irregular Algorithms on GPGPUs Using Fine-Grain Hardware Worklists," in *MICRO*, 2014.

[55] Q. Zhu, B. Akin, H. Sumbul, F. Sadi, J. Hoe, L. Pileggi and F. Franchetti, "A 3D-stacked logic-in-memory accelerator for application-specific data intensive computing," in *3DIC*, 2013.

[56] R. Nasre, M. Burtscher and K. Pingali, "Atomic-free Irregular Computations on GPUs," in *GPGPU-6 at ASPLOS*, 2013.

[57] A. Bakhoda, G. Yuan, W. Fung, H. Wong and T. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *ISPASS*, 2009.

[58] NVIDIA, "NVIDIA Quadro FX 5800," [Online]. Available: http://www.nvidia.com/object/product_quadro_fx_5800_us.html.

[59] K. Rupnow, J. Adriaens, W. Fu and K. Compton, "Performance Metrics for Hybrid Computation in Multi-Tasking Systems," in *FPL*, 2009.

[60] "Predictive Technology Model," [Online]. Available: http://ptm.asu.edu/.

[61] N. S. Kim, T. Kgil, K. Bowman, V. De and T. Mudge, "Total power-optimal pipelining and parallel processing under process variations in nanometer technology," in *Intl. Conf. on CADComputer-Aided Design*, 2005.

[62] "The Rodinia Benchmark Suite, version 2.0," [Online]. Available:

https://www.cs.virginia.edu/~skadron/wiki/rodinia.

[63] J. Lee, V. Sathisha, M. Schulte, K. Compton and N. S. Kim, "Improving Throughput of Power-Constrained GPUs Using Dynamic Voltage/Frequency and Core Scaling," in *PACT*, 2011.

[64] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt and V. J. Reddi, "GPUWattch: enabling energy optimizations in GPGPUs," in *ISCA*, 2013.

[65] C. Gregg, M. Boyer, K. Hazelwood and K. Skadron, "Dynamic Heterogeneous Scheduling Decisions Using Historical Runtime Data," *A4MMC*, 2011.

[66] J. Leskovec and A. Krevl, "Stanford Network Analysis Project (SNAP)," 2014. [Online]. Available: http://snap.stanford.edu/data.

[67] "Bellman–Ford algorithm," [Online]. Available: https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm.

[68] K. Madduri, D. Ediger, K. Jiang, D. A. Bader and D. Chavarria-Miranda, "A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets," in *IPDPS*, 2009.

[69] U. Brandes, "A Faster Algorithm for Betweenness Centrality," in *Journal of Mathematical Sociology*, 2001.

[70] S. A. Manavski, "CUDA compatible GPU as an efficient hardware accelerator for AES cryptography," in *ICSPC*, 2007.

[71] 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*, 2008.

[72] A. Obukhov and A. Kharlamov, "Discrete Cosine Transform for 8x8 Blocks with CUDA," 2008. [Online]. Available: http://developer.download.nvidia.com/compute/DevZone/C/html/C/src/dct8x8/doc/dct8x8.pdf.

[73] NVIDIA, "Image Denoising," 2007. [Online]. Available: http://developer.nvidia.com/cuda-cc-sdk-code-samples#imageDenoising.

[74] Maxime, "Ray Tracing," [Online]. Available: http://www.nvidia.com/cuda.

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

[76] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee and K. Skadron, " Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, 2009.

[77] B. a. Kavinguy, "A Neural Network on GPU," [Online]. Available: http://www.codeproject.com/KB/graphics/GPUNN.aspx.

[78] Microsoft Corp., 2009. [Online]. Available: http://www.microsoft.com/whdc/device/display/wddm timeout.mspx.

[79] A. Cunningham, R. Smith, K. Vättö and J. Walton. [Online]. Available: http://www.anandtech.com/show/5630/indepth-with-the-windows-8-consumer-preview.

[80] L. Luo, M. Wong and W.-m. Hwu, "An Effective GPU Implementation of Breadth-first Search," in *DAC*, 2010.

[81] "Parboil Benchmarks," [Online]. Available: http://impact.crhc.illinois.edu/parboil/parboil.aspx.

[82] W. J. v. d. Laan, "GPU-accelerated Dirac video codec," [Online]. Available: http://diracvideo.org/.