The Pennsylvania State University
The Graduate School

BE(-A)WARE OF DATA MOVEMENT: OPTIMIZING THROUGHPUT PROCESSORS FOR EFFICIENT COMPUTATIONS

A Dissertation in
Computer Science and Engineering
by
Ashutosh Pattnaik

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2019

The dissertation of Ashutosh Pattnaik was reviewed and approved* by the following:

Chita R. Das
Head of the Graduate Program
Distinguished Professor of Computer Science and Engineering
Dissertation Co-Advisor, Co-Chair of Committee

Mahmut T. Kandemir
Professor of Computer Science and Engineering
Dissertation Co-Advisor, Co-Chair of Committee

Anand Sivasubramaniam
Distinguished Professor of Computer Science and Engineering

Prasenjit Mitra
Professor of College of Information Sciences and Technology

Aniruddha Vaidya
GPU Compute Architect, NVIDIA
Special Member

Asit K. Mishra
Senior Deep Learning Computer Architect, NVIDIA
Special Member

*Signatures are on file in the Graduate School.

Abstract

General-Purpose Graphics Processing Units (GPGPUs) have become a dominant computing paradigm to accelerate diverse classes of applications primarily because of their higher throughput and better energy efficiency compared to CPUs. Moreover, GPU performance has been rapidly increasing due to technology scaling, increased core count and larger GPU cores. This has made GPUs an ideal substrate for building high performance, energy efficient computing systems. However, in spite of many architectural innovations in designing state-of-the-art GPUs, their deliverable performance falls far short of the achievable performance due to several issues. One of the major impediments to improving performance and energy efficiency of GPUs further is the overheads associated with data movement. The main motivation behind the dissertation is to investigate techniques to mitigate the effects of data movement towards performance on throughput architectures. It consists of three main components. The first part of this dissertation focuses on developing intelligent compute scheduling techniques for GPU architectures with support for processing in memory (PIM) capability. It performs an in-depth kernel-level analysis of GPU applications and develops prediction model for efficient compute scheduling and management between the GPU and the processing in memory enabled memory. The second part of this dissertation focuses on reducing the on-chip data movement footprint via efficient near data computing mechanisms. It identifies the basic forms of instructions that are ideal candidates for offloading and provides the necessary compiler and hardware support to enable offloading computations closer to where the data resides for improving the performance and energy-efficiency. The third part of this dissertation focuses on investigating new warp formation and scheduling mechanisms for GPUs. It identifies code regions that leads to the under-utilization of the GPU core. Specifically, it tackles the challenges of control-flow and memory divergence by generating new warps dynamically and efficiently scheduling them to maximize the consumption of data from divergent memory operations. All the three techniques independently and collectively can significantly improve the performance of GPUs.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I deeply appreciate and thank the many people who provided intellectual contributions and emotional support. Above all, I thank my advisor, Chita Das, for his guidance, support, and encouragement throughout my Ph.D. tenure. He has always inspired me to be a better researcher and never settle for anything less than the best. I have learnt a lot about problem-solving, technical writing and many other important aspects of being a good researcher from him. I have yet to come across a more positive and encouraging person in my life. I am highly indebted to him for all the effort he has put into making me a researcher and a patient human being. I also thank his entire family, especially Namita Das, for their unconditional support.

I would like to thank my dissertation co-advisor, Mahmut Kandemir, who has been a constant source of inspiration and strength. He has taught me the value of hard work and patience, and instilled in me a no-give up attitude. The brainstorming sessions with him have always helped me think from various different aspects, adding a level of completeness to the work.

I also deeply thank my co-advisor, Anand Sivasubramaniam, who has been a constant source of creativeness and innovation to me. He has always championed for higher quality and his insights have helped the work in this dissertation achieve a wider appeal and impact. The discussions with him have helped me widen my horizon and changed my problem solving thought process. His insistence on knowing the prior work well has been one of the most useful skills I have learnt during my Ph.D. tenure.

I am grateful to Prasenjit Mitra, Aniruddha Vaidya and Asit Mishra for serving on my dissertation committee. I am also thankful to Onur Mutlu for his interest and help on my paper. I am grateful to John "Jack" Sampson for the numerous technical (and non-technical) discussions that have helped me articulate my research efficiently.

My PhD journey has been one of the most wonderful (and stressful) periods of my life. Apart from my advisors and committee members, this dissertation would not have been possible without the immense support of Adwait Jog, Onur Kayıran and Nachiappan Chidambaram. From fixing references, creating figures to modifying simulators and writing

papers, they have taught me all. I will always be indebted to them. Moreover, I thank Asit Mishra and Bikash Sharma, who have been invaluable in sharing their technical and non-technical life experiences with me. They have helped me make tough decisions through multiple crossroads during my Ph.D. tenure. And special thanks to Gauravi Dudhbhate for her immense support and encouragement.

I also thank my peers and great friends Prashanth Thinakaran, Iyswarya Narayanan, Prasanna Rengasamy, Tulika Parija, Aditi Bhat, Jashwant Gunasekaran, Xulong Tang, Haibo Zhang, Anup Sarma and Sonali Singh. I have learnt a great deal about other research areas from them and it has expanded my knowledge greatly. They have been amazing friends who have made my life at Penn State a little bit easier. I thank Kashyap Dixit, Tuba Kesten, Sampad Mohapatra, Sandeepa Bhuyan, Huaipan Jiang, Cyan Mishra, Vivek Bhasi, Siddhartha Rai, Jihyun Ryuu, Shruti Mohanty, Srivatsa RS, Sumitha George and Balakrishnan Swaminathan for making life at Penn State enjoyable. I am very thankful to my friends Adyasha Mishra, Sanjeeb Panda, Garima Chopra and Pritesh Kanani for their emotional support and motivation.

My internship experiences at AMD Austin and AMD Sunnyvale provided immense industry and research experience. I am thankful to Joseph Greathouse, Nuwan Jayasena and Yasuko Eckert from AMD for providing me with internship opportunities.

Finally, I thank my father, Pradeep Kumar Pattnaik, who has been a role model to me and encouraged me to be always better; my mother, Anuradha Pattnaik, who has given me unconditional support and love; my sister, Prachi Pattnaik, and my brother-in-law, Pravin Kumar, for being a constant source of encouragement and support.

Chapter 2 contains material from "Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities", by Ashutosh Pattnaik, Xulong Tang, Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu and Chita R. Das, which appears in the *Proceedings of the 25th Parallel Architecture and Compilation Techniques (PACT) 2016*. The dissertation author is the primary investigator and author of this paper. The material in Chapter 2 is copyright by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Chapter 3 contains material from "Opportunistic Computing in GPU Architectures", by Ashutosh Pattnaik, Xulong Tang, Onur Kayiran, Adwait Jog, Asit Mishra, Mahmut T. Kandemir, Anand Sivasubramaniam and Chita R. Das which appears in the *Proceedings of the 46th International Symposium on Computer Architecture (ISCA) 2019*. The

Thank you all.

Dedication

To my dear parents and sister for being a constant source of great support and inspiration.

Chapter 1

Introduction

In recent times, there has been a push for building faster and energy-efficient computing systems. This is mainly due to the economy heading towards green computing, which requires computations to become cheaper overall accounting from capital expenditure to running costs. In United States, an executive order was passed, whose primary goal is to build energy-efficient high performance computing systems [2]. Specifically, the overarching goal is to achieve exascale throughput with a power budget of 20 MW [3]. An ideal processing unit for such systems should have the following three key characteristics: (i) flexibility in programming; (ii) high throughput; and (iii) high energy-efficiency. There are many different types of processing elements ranging from central processing units (CPUs), general purpose graphics processing units (GPGPUs), field programmable gate arrays (FPGAs) to domain specific accelerators, but only GPUs[1] try to maximize all the three key characteristics, while other processing elements maximize, at best, only two of the three characteristics. Subsequently, GPUs have been used to accelerate applications from multiple domains such as graphics and computer vision [4], medicine [5–8], scientific computing [9], finance [10, 10–12] and social media [13] due to their highly data-parallel computation model that allows for orders of magnitude improved performance and

[1] We use the term GPGPUs and GPUs interchangeably in this dissertation.

energy-efficiency when compared to CPUs [14]. This has led to their widespread adoption in many high performance computing systems as seen on the Top500 [15] and Green500 [16] lists. Therefore, GPGPUs are likely to play a promising role in achieving the exascale target. On the other side of the spectrum, GPGPUs are now heavily being used on energy limited devices such as wearables [17], smart phones [18], Internet-of-Things (IoTs) [19] and autonomous cars [20], where the size of the raw data that needs to be processed has exploded exponentially [21]. Furthermore, the interactive/real-time nature of these devices has made it infeasible to completely offload them to cloud without any local processing [22, 23]. Therefore, high performance, energy-efficient GPUs are needed from wearable computing to warehouse computing.

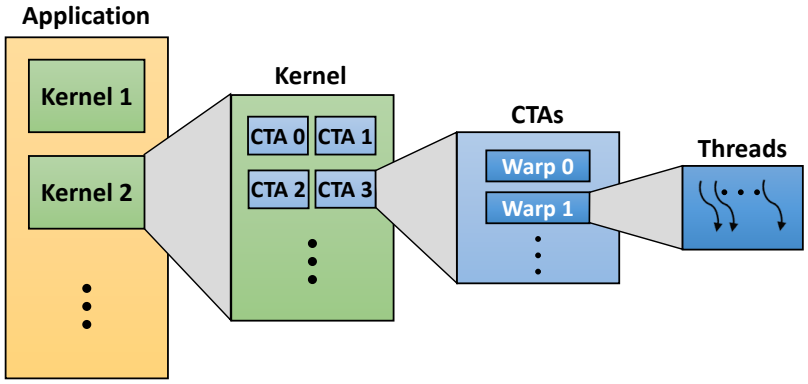


Figure 1.1: A typical GPGPU application hierarchy.

1.1 Background

A typical GPGPU consists of many programmable compute units that host thousands of concurrently executing threads. Since the inception of GPGPUs, there are multiple programming models such as CUDA [14] and OpenCL [24] that have been developed to reduce the programmability burden on explicitly writing parallel codes for GPGPUs. GPGPU applications are divided into multiple hierarchy of groups of threads. Figure 1.1 shows a typical GPGPU application's hierarchy. Usually, a GPGPU application is made

up of many kernels, which in turn consists of multiple threads. A *kernel* is the smallest unit of work that a host (CPU) can offload to a GPGPU for execution. Kernels are further divided into *thread blocks* or *Cooperative Thread Arrays* (CTAs) that are the smallest schedulable unit for a compute unit present inside a GPGPU. Finally, a CTA is broken into a set of contiguous threads (usually at a granularity of 32 threads) known as *warp* or *wavefront*. Typically, all the threads within a warp run in lock step inside a compute unit while sharing the same instruction stream. In a typical GPGPU architecture (shown in Figure 1.2), each compute unit has it's respective warp scheduler, private L1 data and instruction caches along with multiple specialized caches. The compute units are connected to a interconnect that links them to the shared, partitioned last-level caches (LLC), which in turn are connected to the memory controllers and DRAM. Therefore, any communication past the private caches, either to the LLC or DRAM, needs to traverse the interconnect. The current state-of-the-art GPU architectures such as NVIDIA Volta [25] and AMD Vega [26] have up to 80 and 64 GPU cores, respectively. The last-level caches of these GPUs can be as big as 4.5 MB and the memory bandwidth can be as high as 1 TB/s.

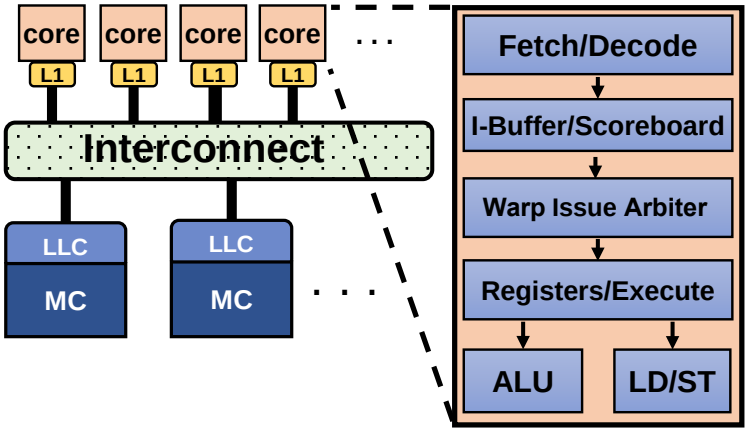


Figure 1.2: A typical GPGPU architecture.

There has been considerable progress in making GPUs more amenable towards general purpose applications. For example, NVIDIA and AMD have added new features such as

Dynamic Parallelism [27, 28], which enables the GPU to create and launch new kernels on-the-fly without the need of a host device intervention. They have also added support for variable-precision floating point arithmetic to improve throughput of applications with low precision requirements. They have also added support for atomic operations, unified memory and multi-application execution [25, 26]. NVIDIA has gone a step further and added tensor cores into their GPU datapath to enable efficient tensor operations that are needed for most machine learning applications [25]. Moreover, many new architectures that make use of GPUs have been developed such as heterogeneous CPU-GPU architectures [29], interposer-based GPU systems [30] and multi-chip module GPUs [31]. These features and designs have pushed the definition of what GPUs were supposed to be - a *coprocessor*, and ushered GPUs into the domain of first class computing engines that are as capable as general purpose CPUs.

Historically, GPU architectures can be traced back to era of video graphics array, where most components were fixed function units. They evolved over time by adding more and more functionality and the current generation state-of-the-art GPUs are composed of many scalable parallel processors [25, 26, 32]. GPUs are akin to vector processors in the sense that they perform vector operations rather than scalar operations. However, GPUs have three main characteristics that differentiate them from vector processors [33]. First, GPUs support the concept of multi-threading where a warp can be context switched out for another ready warp in case of stalls to hide long latency operations. Second, GPUs have specialized components in their pipeline such as software-managed scratchpad, constant cache, texture cache, tensor cores, etc. in order to maximize the performance on a variety of applications. Finally, GPUs are able to support irregular memory operations at a much finer granularity than traditional vector processors which deal with only vector data structures. On the other hand, certain GPU architectures have also evolved from VLIW based architectures [34] such as AMD Graphics Core Next (GCN) architecture [35]. Similar to a VLIW machine, the GCN architecture has 4 separate SIMD units, with each having 16-wide lanes. Potentially, 4 different instructions can run across the 4 SIMD units, while the 16 lanes execute the

same instruction. All the optimizations proposed in this dissertation are not exclusive to any specific evolution of the GPU architecture and can be easily extended to any other GPU architectures with minimal changes. GPUs can also be traced back to stream processors [36, 37]. Stream processors were designed for media processing applications and brought in the concepts of organizing an application into streams and kernels. Current GPU applications still make use of this organization and have added more hierarchy (thread blocks and warps) in the application organization to maximize the utilization of GPUs.

1.2 The Problem

GPUs were originally designed to run highly parallel and streaming graphics applications, but recently, with the advent of GPGPUs, more and more irregular applications have been ported to GPUs which has exposed the inherent bottlenecks of the GPU architecture, and therefore, has limited the potential scalability prospects. Specifically, even though GPUs have highly energy-efficient compute units and high peak throughput, they still suffer from idle cycles and an inefficient memory system, leading to underutilized resources and sub-optimal performance and energy-efficiency. With technology scaling, the relative energy consumption for moving data is orders of magnitude more than that of a compute operation [38], and therefore, any application with large data movement energy costs hinders the scalability of GPGPUs. The data movement overheads also translate to performance degradation as the major portion of the idle cycles are primarily due to the inability of the GPU to hide the long latency memory requests. Although GPUs are touted as *latency insensitive*, for certain classes of applications, the long latencies are exposed and become a performance bottleneck mainly due to two reasons: (1) the large distances that the data needs to traverse to reach the compute units (either off-chip or on-chip), and (2) ineffective compute and memory request scheduling that increases data movement leading to even longer latencies. Therefore, the continuous scaling of performance and energy-efficiency of GPUs is a non-trivial task, with the biggest

impediment being the memory system energy consumption and long latencies due to data transfer overhead [39].

1.3 Contributions

This dissertation addresses the above issues by focusing on the problem of data movement in throughput processors. In this context, the dissertation proposes techniques that are employed at the *core*, *caches* and *memory*. First, this dissertation shows that by intelligently picking and scheduling kernels, either at the core or memory can significantly improve performance and energy efficiency. Second, this dissertation shows that by offloading certain instructions from the core to the caches and other cores can help improve performance and energy efficiency as well. Finally, this dissertation shows that by intelligently generating and scheduling warps at the core, it is possible to improve core utilization and reduce memory stalls for increased performance.

Contribution I: Scheduling Techniques for Processing In Memory Enabled Throughput Processors [40]. The first contribution of this dissertation is a novel kernel scheduling and offloading mechanism that targets to reduce the off-chip communication between the GPU cores and the memory system, while improving performance. The end goal of this is to develop mechanisms to fully and automatically exploit the potential of GPGPU architectures comprising of processing-in-memory (PIM) units in their main memory. PIM units have access to high bandwidth and lower data movement but limited compute capability, and therefore, by integrating PIM units to the GPU, it creates a heterogeneous architecture with two processing engines having different compute and bandwidth characteristics. To this end, two key code scheduling problems are investigated. First, we investigated how to automatically identify the code segments that should be offloaded to the PIM units in memory. Second, we examined how to concurrently schedule multiple kernels on the main GPU and the PIM units. For this, we

first characterize the execution behavior of different applications at the kernel granularity to estimate the performance and energy benefits when each individual kernel is placed in the main GPU or PIM units. Based on this insight, it proposes two new runtime techniques to address the two scheduling problems.

Contribution II: Enabling Opportunistic Computations on Throughput Processors for Reduced On-Chip Data Movement [41]. In order to minimize the data movement overhead stated earlier, we propose techniques that target to reduce the on-chip communication between the GPU cores and the last-level caches. As GPUs keep scaling, their dies are becoming more and more larger, and therefore, the data needs to travel even farther than before leading to higher energy consumption and longer latencies. This work considers reducing on-chip data movement in GPUs by opportunistically offloading computations either to (1) the last-level-caches, or (2) the ALU of another GPU core, which would result in the lowest on-chip data movement for that computation. For applications that are highly streaming in nature or are sensitive to the LLC, offloading will help reduce the data movement costs and avoid bringing in data into the private caches that are not reused appropriately. To this end, it investigates two offloading mechanisms (LLC-Compute and Omni-Compute) that, at runtime, dynamically identifies the instruction sequences to offload, finds an ideal node to execute the instruction sequence that will lead to minimal data movement of the operands and then offloads it to the desired node for computation.

Contribution III: Design and Analysis of Control-Flow and Memory Divergence-aware Scheduling in Throughput Processors. The third contribution of this dissertation identifies the key bottlenecks in the compute and memory request generation pipelines. It finds that in modern GPGPUs architectures, due to their nature of executing a group of threads, *warps*, in lockstep can lead to inefficient execution for applications with compute or memory access irregularity. This inefficiency arises due to two main reasons: (1) control-flow divergence; and (2) memory divergence. These

divergences can hamper performance by reducing the functional unit utilization and also lead to excessive data movement.

For this, it first conducts a detailed characterization of control-flow and memory divergence that is present in applications. Using this knowledge, it develops a software-assisted hardware mechanism, called Shadow Engines, that at runtime can mitigate the effects of control-flow divergence and memory divergence via formation of new warps and efficient warp scheduling techniques.

Dissertation Organization: This dissertation is organized into five chapters. Chapter 2 presents kernel scheduling and runtime management techniques for a processor in memory enabled GPU architecture. Chapter 3 develops two near data computing mechanisms, called LLC-Compute and Omni-Compute, to reduce on-chip data movement for improved performance and energy efficiency. Chapter 4 introduces Shadow Engines, a dynamic GPU warp management mechanism to reduce control-flow and memory divergence in the GPU pipeline for increased performance. Chapter 5 provides conclusions, a summary of key results and discusses the potential future research aspects in throughput processors.

Chapter 2

Scheduling Techniques for Processing In Memory Enabled Throughput Processors

Processing data in or near memory (PIM), as opposed to in conventional computational units in a processor, can greatly alleviate the performance and energy penalties of data transfers from/to main memory. Graphics Processing Unit (GPU) architectures and applications, where main memory bandwidth is a critical bottleneck, can benefit from the use of PIM. To this end, an application should be properly partitioned and scheduled to execute on either the main, powerful GPU cores that are far away from memory or the auxiliary, simple GPU cores that are close to memory (e.g., in the logic layer of 3D-stacked DRAM). This chapter investigates two key code scheduling issues in such a GPU architecture that has PIM capabilities, to maximize performance and energy-efficiency.

2.1 Introduction

Graphics Processing Units (GPUs) provide very high computational bandwidth at a competitive power budget. These characteristics have led to their deployment in a wide range of platforms, including in many machines that appear in the Top500 and Green500 lists [15, 16]. Although GPUs are likely to play a promising role in the design of exascale systems, continuous scaling of their performance and energy efficiency will not be an easy task. One of the biggest impediments to this continuous scaling is the memory system energy consumption due to data transfer overhead [39]. A typical 64-bit DRAM access consumes about 100-1000X the energy consumed by a double-precision floating point operation [38, 39], and this gap could increase with technology scaling [42]. Even with optimistic assumptions about improvements in memory technology, reducing the total DRAM access energy from approximately 18-22 pJ/b (in modern GDDR5) to 4 pJ/b and sustaining it for over 100,000 nodes, memory can still consume a significant fraction (e.g., 70%) of the total system's power budget [30].

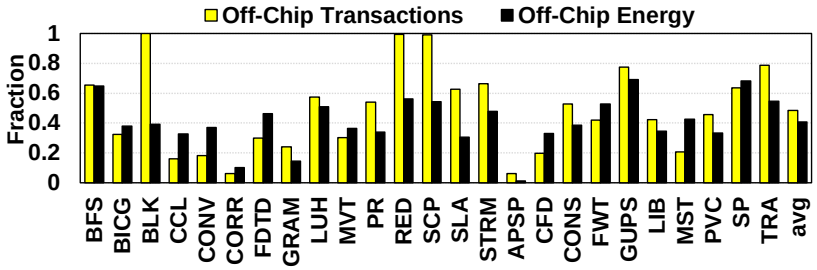


Figure 2.1: Data movement and system energy consumption caused by off-chip memory accesses.

To examine the data movement overhead, we simulate a state-of-the-art GPU system and analyze 25 GPU applications (Section 2.6 provides our experimental methodology). Figure 2.1 illustrates the data movement and energy consumption overheads of transferring data between memory and computational units across the evaluated applications, by showing: (1) the fraction of all data movement in the system that is due

to off-chip transactions between memory and the GPU, and (2) the fraction of total system energy consumption that is due to this off-chip data movement. We observe that memory accesses result in 49% of all data movement and are responsible for 41% of the energy consumption of the system.

Figure 2.2 shows the performance loss due to the overhead of transferring data from/to main memory, by plotting the normalized performance of our applications compared to an *idealized* system, where all off-chip memory requests in our baseline are *magically* eliminated, i.e., forced to hit in the last-level cache.[1] Averaged across 25 applications, main memory accesses lead to 45% performance degradation.

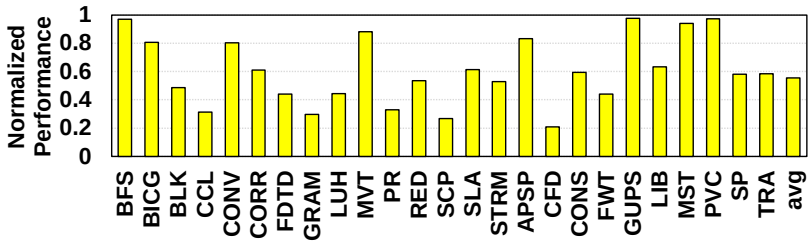


Figure 2.2: Performance normalized to a hypothetical GPU where all the off-chip accesses hit in the last-level cache.

A promising approach to minimize data movement, energy and performance overheads of main memory accesses is to move memory-intensive computations closer to memory, i.e., Processing-In Memory (PIM) [60–63], also known as Processing-Near Memory (PNM) or Near-Data Computing (NDC) [64]. The core of the PIM concept is to have computational units that are closely integrated with memory such that data can be moved from memory to those units at much higher bandwidth, lower latency and lower energy than doable from memory to the main processor. While the PIM concept goes back to late 1960s [65] and it had gathered some momentum through several projects in 1990s (e.g. [60, 66–70]), the main bottleneck in widespread adoption of PIM has been the technological limitation of integrating computational units very close to memory. With the significant advances in

[1]The minimum DRAM latency after the last-level cache is 100 cycles (see Section 2.6 for details). The average latency is higher due to significant contention observed in the DRAM system [43–59].

adoption of 3D-stacked memory technology that tightly combines a logic layer and DRAM layers [62, 63, 71–75], this limitation has been overcome and PIM has become a likely-viable approach to improve system design.

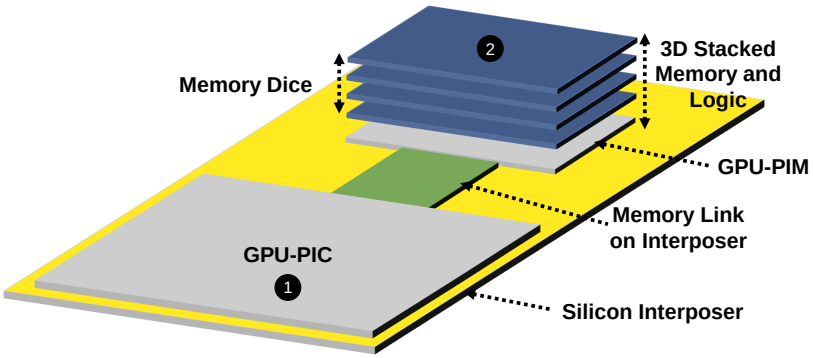


Figure 2.3: A PIM-assisted GPU Architecture. GPU-PIC is a traditional GPU that is connected to the 3D stacked memory via I/O links on the silicon interposer. GPU-PIM is a relatively smaller GPU (same ISA as GPU-PIC but lower compute throughput) placed under the 3D stacked memory that has access to very high bandwidth compared to GPU-PIC.

The PIM approach has recently been explored (e.g., [30, 76]) for reducing memory bandwidth and minimizing the data transfer overheads between GPU cores and off-chip DRAM. A promising way to integrate PIM to a GPU-based system is what we call a *PIM-Assisted GPU architecture*, where at least one 3D-stacked memory chip is placed adjacent to the GPU chip, and both chips are interconnected via a memory link on an interposer (as depicted in Figure 2.3).[2] The 3D-stacked memory contains a base logic layer, housing GPU cores. This architecture has two types of compute engines: (1) large and powerful primary GPU cores, called GPU-PIC, i.e., processing-in-core, which are similar to modern GPU cores, and (2) smaller and less powerful GPU cores, called GPU-PIM, which are placed in the logic layer under main memory and assist in performing computation. No prior work explored how to fully exploit this architecture such that appropriate parts of an application are identified and scheduled to utilize *both* the main GPU cores (GPU-PIC) and cores in memory (GPU-PIM) to maximize the

[2]Section 2.2.2 discusses details and advantages of this architecture.

performance and energy-efficiency of the entire system.

The goal in this chapter is to develop mechanisms to fully and automatically exploit the performance and energy-efficiency potential of PIM-Assisted GPU architectures. To this end, we investigate **two key code scheduling problems**. First, *how to automatically identify the code segments to be offloaded to the GPU cores in memory.* Second, *how to concurrently schedule multiple kernels on the main GPU and the cores in memory.* To address these problems, one must consider several questions such as (1) what the granularity of the code segment that is offloaded to GPU-PIM should be? (2) how to determine which code segments benefit from being executed on GPU-PIM? (3) how to efficiently distribute work between the main GPU and the PIM engine to maximize system performance while executing each code segment on its preferred cores as much as possible?

In order to answer the above questions, we first characterize the execution behavior of different applications at the **kernel granularity** to estimate the performance and energy benefits when each individual kernel is placed in the main GPU or GPU-PIM. Because the CPU offloads computation to the GPU at the kernel-granularity, maintaining the same granularity for PIM execution enables low-overhead computation offloading from the CPU to both GPU-PIC and GPU-PIM. Based on this insight, we propose **two new runtime techniques**, the primary contributions of this chapter, that address the two scheduling problems.

Kernel Offloading. As an application exhibits varying computation and memory demands during different phases of execution, some kernels (e.g., those that fit in the main GPU scratchpad) benefit more from executing on the main GPU, GPU-PIC, and others (e.g., those that overwhelm the memory bandwidth to the main GPU) on the GPU in memory, GPU-PIM. Thus, identifying the affinity of the kernels correctly and scheduling each on the appropriate computational engine would improve performance. To solve this problem, we develop a *regression-based affinity prediction model and mechanism* that accurately identifies, at runtime, which kernels would benefit from executing on PIM cores

and offloads them to the GPU cores in memory. Our regression model, which is built on an in-depth kernel level analysis, considers three broad kernel-level metrics (memory intensity, kernel parallelism, and shared memory (software managed scratchpad) intensity) and is trained using applications randomly picked from our pool of 25 applications (i.e., the training set). Our detailed experimental evaluations on the remaining applications (i.e., the test set) show that the proposed mechanism improves average performance by 25% and energy efficiency by 28% compared to a baseline conventional GPU architecture that contains the same number of GPU cores as that of the combined number of GPU cores present in both GPU-PIC and GPU-PIM.

Concurrent Kernel Management. We find that even a highly accurate prediction mechanism for kernel offloading can leave many resources (e.g., GPU-PIC) underutilized. During the execution of a kernel on either the main GPU or GPU-PIM, the other device is left unutilized, which limits achievable system performance. Thus, identifying *independent kernels* that can be scheduled concurrently and scheduling them on GPU-PIC and GPU-PIM at the same time, in a manner that minimizes overall application execution time, can significantly improve system performance and efficiency. To solve this problem, we develop a *concurrent kernel management* mechanism that uses the affinity prediction model, a new regression-based kernel execution time prediction model, and dependency information across kernels to decide which kernels to schedule concurrently on the main GPU cores and the GPU cores in memory. Our detailed experimental evaluations indicate that our technique improves average performance by 42% and energy-efficiency by 27%, compared to the same baseline described above.

This work makes the following major **contributions**:

- It provides an in-depth kernel-level analysis of GPU application behavior with respect to suitability for Processing in Memory. It makes an experimentally-supported case for the use of *kernel granularity* to identify and schedule GPU program code segments to execute on either the main GPU cores or the GPU cores in memory.

- It develops a new regression-based affinity prediction model to estimate the best compute engine to execute a kernel in a PIM-assisted GPU architecture. We show how to use this model to guide a new *kernel offloading* mechanism to GPU cores in memory.

- It proposes a new execution time prediction model for kernel execution in a PIM-assisted GPU architecture. We show how to use this model to guide a new *concurrent kernel management* mechanism that executes multiple kernels concurrently in the main GPU and the in-memory GPU cores in a PIM-assisted GPU architecture.

- We comprehensively evaluate the *kernel offloading* and *concurrent kernel management* mechanisms and show that both performance and energy efficiency significantly improve while requiring no support from the programmer.

2.2 Background

This section provides a brief background on GPUs and the PIM-assisted GPU architectures.

2.2.1 Conventional GPU Architectures

Recall from Figure 1.2, our baseline conventional GPU consists of multiple cores, also called streaming-multiprocessors (SMs)[3] in NVIDIA terminology [77] or compute units (CUs) in AMD terminology [26]. Each SM has a private L1 data cache, a texture cache and a constant cache, along with a software-managed scratchpad memory (called *shared memory*). The SMs are connected to memory channels (partitions) via an interconnection network. Each memory partition is directly connected to a partition of the L2 cache, which is shared by all the cores, and a memory controller. The memory requests are buffered and scheduled by the memory controllers [54, 78, 79]. There are multiple memory controllers, each controlling

[3]In this dissertation, we use the terms *core*, *SM* and *CU* interchangeably.

a memory partition. Data is interleaved at the chunk granularity across the controllers. The parallel parts of a CUDA/OpenCL application, which are offloaded to the GPU, are called *kernels*. The execution of an application starts with the launch of a kernel on the GPU. The kernel launch involves copying the kernel code and the data from the CPU memory to the GPU memory. Once the kernel finishes execution on the GPU, its results are copied back to the CPU memory from the GPU memory.

2.2.2 PIM-Assisted GPU Architectures

As we discussed in Section 2.1, 3D-stacked memory technology enables the ability to place computational units in the base logic layer that is underneath the memory stacks [62,63,71–75]. There can be multiple strategies to exploit such a PIM technology in a GPU system.

One strategy is to stack the memory directly on top of a conventional GPU architecture, by placing a conventional GPU in the logic layer [80]. Although such an organization provides the benefits of tight GPU and memory coupling such as high bandwidth and low access latency, it has two primary issues. First, heat from the processor might significantly degrade the retention time of the 3D-stacked memory [81]. Consequently, the refresh rate of the DRAM might need to be increased, leading to reduced peak performance and higher energy consumption [81–85]. Second, such an organization limits the total memory capacity that could be stacked within the area of the processor.

Another strategy [30, 76] to exploit PIM in a GPU system is to keep the main GPU same as in conventional systems but connect to it, via memory links on a silicon interposer, one or more 3D-stacked memory units that are capable of doing computation, as depicted in Figure 2.3 with one 3D-stacked memory. The base logic layer of each 3D-stacked memory houses an auxiliary GPU that is simpler and more power-efficient than the main GPU. Such an organization has considerably lower thermal constraints than the previous strategy and is more scalable in terms of memory capacity. However, if

computations are not scheduled appropriately across the main GPU and the computational units in the 3D-stacked memories, the energy consumption, latency and bandwidth overheads of the interposer may limit performance due to excessive communication between the main GPU and the 3D-memory stacks.

In this chapter, we call the latter organization (of Figure 2.3) the *PIM-Assisted GPU architecture*, and aim to maximize its benefits by scheduling computations intelligently across the main GPU and PIM units. This architecture is a heterogeneous system. The main GPU chip, which we call the processing-in-core architecture (GPU-PIC ❶ in Figure 2.3), provides high throughput to compute-intensive GPGPU applications, but it has limited memory bandwidth due to its horizontal integration with the memory stack (bottlenecked by the memory links). On the other hand, the PIM cores on the base logic layer of the 3D memory stack, which we call the processing-in-memory architecture (GPU-PIM ❷ in Figure 2.3), achieves the full bandwidth and energy efficiency of 3D stacking of memory and logic, but provides a peak instruction throughput lower than that of GPU-PIC due to the smaller number of the execution engines in the logic layer. Therefore, placing computation correctly on these two different types of GPU units, GPU-PIC and GPU-PIM, is critical for performance and energy efficiency. For example, computations that are memory-intensive and can tolerate the lower parallelism present in the logic layer of GPUs are likely better executed on GPU-PIM instead of GPU-PIC.

Unlike GPU-PIC, GPU-PIM has a direct interface to 3D-stacked DRAM. Therefore, GPU-PIM is able to sustain much higher memory bandwidth (in our configuration, 4×) and it experiences much lower memory latency than GPU-PIC. Because of this, we observe that GPU-PIM does *not* significantly benefit from having an L2 cache and we evaluate a GPU-PIC design without an L2 cache. Table 2.4 provides the details of our GPU-PIC and GPU-PIM configuration, which is similar to the TOP-PIM configuration [30]. We also perform a sensitivity analysis in Section 2.8 on the number of cores and cache in GPU-PIM. Note that the cores in GPU-PIC and GPU-PIM use the same ISA, and thus, have the same

programmability features.

Thermal Feasibility. Eckert et al. [86] provide extensive models to demonstrate the thermal feasibility of PIM-based GPU architectures. They argue that the power consumed by the logic layer at the GPU-PIM at an ambient temperature of 30°C can be as much as 50W and describe that this is thermally feasible. Considering their study and configuration parameters from [30], our GPU-PIM has four times fewer SIMD compute units, similar to [76]. We estimate the maximum chip power usage of GPU-PIM when running MaxFlops [87][4] to be approximately 45W, using GPUWattch [88]. GPUWattch models NVIDIA Fermi [77] SMs, which are obsolete and not power-efficient compared to current generation GPU cores [89]. Therefore, we believe GPU-PIM will be even more power efficient when fabricated for state-of-the-art and future power-efficient designs.

2.3 Motivation

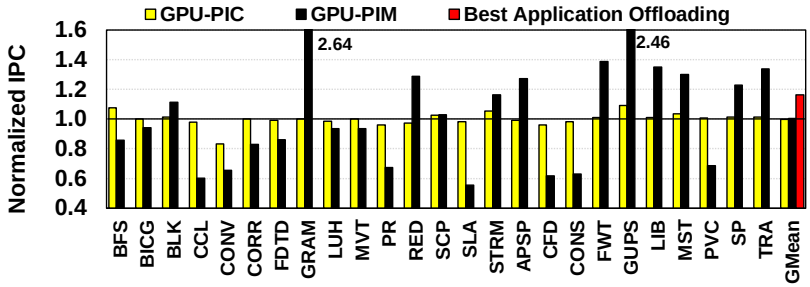
The PIM-assisted GPU architecture is a scalable and heterogeneous substrate. It provides the flexibility of adding more computational units on the GPU-PIC, and more memory capacity and bandwidth by incorporating additional 3D memory stacks containing GPU-PIM. At the same time, GPU-PIC and GPU-PIM architectures are quite different in terms of their ability to cater to applications with varying computation and memory demands. Given such heterogeneity in the PIM-assisted GPU architecture, it might be desirable for the CPU to offload a compute-bound GPGPU application onto GPU-PIC and a memory-bound GPGPU application onto GPU-PIM. However, this is not optimal, as we demonstrate in this section since it does not fully utilize both the GPU-PIC and GPU-PIM. There are many challenges in designing an offloading strategy to *maximize overall application performance and energy efficiency* by appropriately partitioning and scheduling an application across GPU-PIC and GPU-PIM. This section motivates the potential opportunities and limitations

[4]MaxFlops is compute-intensive benchmark that exhibits high dynamic core power consumption.

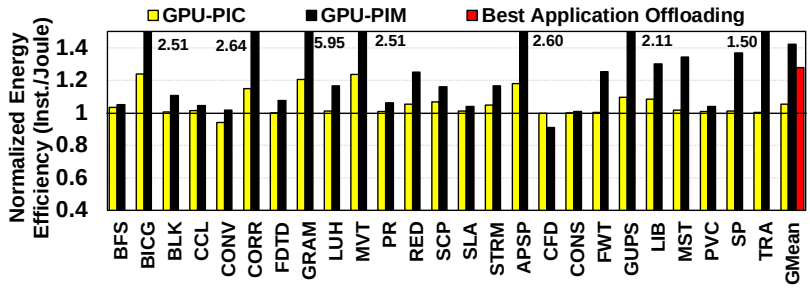
of *application offloading* to motivate our approach of *kernel offloading*.

2.3.1 Benefits of Application Offloading

Figure 2.4 shows the normalized performance (in terms of IPC) and energy efficiency (in terms of Instructions/Joules), compared to our baseline conventional GPU architecture with equivalent number of computation units (40 cores), when each application from our workload suite are offloaded by the CPU on either the GPU-PIM (8 cores) or the GPU-PIC (32 cores) in our PIM-Assisted GPU Architecture. *Best Application Offloading* shows the average performance and energy efficiency when each application is, with ideal knowledge, offloaded to the computation unit that provides the *best performance* for that application. We make three major observations.



(a) Performance.



(b) Energy Efficiency.

Figure 2.4: Effect of application offloading.[5]

[5]The results are normalized to a conventional GPU which has the combined peak instruction throughput of GPU-PIC and GPU-PIM (i.e., 40 cores). The entire application is offloaded to either

First, offloading all applications to either GPU-PIC or GPU-PIM yields similar average performance because some applications prefer GPU-PIC and others GPU-PIM. We find that many applications significantly gain from the high memory bandwidth and low memory latency benefits of GPU-PIM. Some applications, e.g., `GRAM` and `APSP`, benefit from GPU-PIM due to low memory access latencies. These two applications (`GRAM` and `APSP`) have limited parallelism (as shown later in Table 2.2), leading to poor latency tolerance. Neither of these applications stress the memory bandwidth and they have a small fraction of accesses going to off-chip memory (Figure 2.1). In contrast, many applications, such as `CCL`, `PR`, and `CFD`, experience performance degradation when executed on GPU-PIM, because they 1) are bottlenecked by the limited computational power of the PIM cores, 2) do not effectively utilize the high bandwidth of GPU-PIM because they generate a small number of memory requests.

Second, offloading all applications to GPU-PIM provides the highest average energy efficiency (Instructions/Joule), even higher than the *Best Application Offloading* mechanism which chooses the unit that provides the best *performance* on a per-application basis. This is because GPU-PIM leads to a much more energy-efficient memory system, on average. Applications such as `GUPS`, `GRAM`, and `APSP` benefit significantly from GPU-PIM in terms of energy efficiency, primarily because of the reduced execution time due to the performance improvements.

Third, an optimal application offloading scheme that can detect the best platform to execute an application on, based on performance, can provide both performance and energy improvements than offloading always to either GPU-PIC or GPU-PIM. On average, GPU-PIM improves energy efficiency by 42% over GPU-PIC while having the same performance of GPU-PIC. The optimal application offloading scheme (in terms of performance) improves performance by 16% and energy efficiency by 28% over the baseline. Note that the optimal scheme is optimized for performance, not energy efficiency, and therefore its energy efficiency

GPU-PIC (32 cores) or GPU-PIM (8 cores).

is less than that of offloading all applications to GPU-PIM.

We also note that an optimal offloading scheme could be different based on the metric to optimize for. For example, for `MVT`, GPU-PIM is much more energy-efficient but lower performance than GPU-PIC. There is no clear winner as one could optimize for either performance or energy-efficiency, depending on the use-case. In this work, we focus on optimizing performance. However, we also demonstrate the positive impact of our schemes on energy efficiency.

2.3.2 Limitations of Application Offloading

We demonstrate the limitations of application offloading. Even for an optimal *application* offloading strategy, we find two major limitations that need to be addressed to make offloading more efficient in terms of performance.

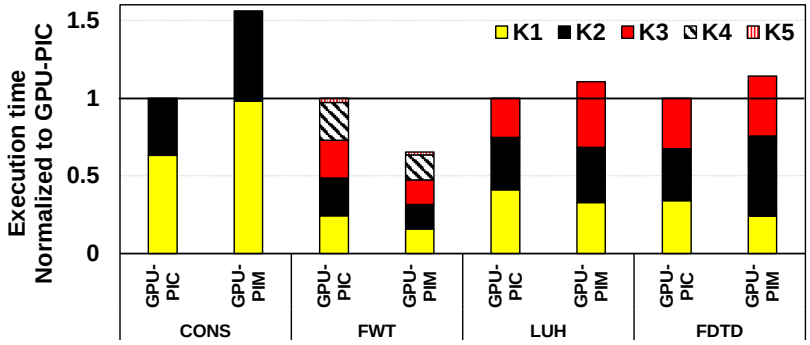


Figure 2.5: Breakdown of the execution time across different kernels for four representative GPGPU applications.

Limitation I: Lack of Fine-Grained Offloading. We observe that offloading at the granularity of each application is too coarse-grained to take advantage of the different characteristics of code present within an application that can favor either GPU-PIC or GPU-PIM. To motivate this, Figure 2.5 shows a *kernel-level*, i.e., finer-grained, execution time breakdown of four representative GPGPU applications on GPU-PIC and GPU-PIM, normalized to the execution time on GPU-PIC. We show only the kernels that dominate

each application's execution time. Three observations are in order. First, `CONS` has two representative kernels and execution times of both kernels increase on GPU-PIM. Therefore, GPU-PIM is not an appropriate computation unit for any of `CONS`'s kernels. Second, `FWT` has four representative kernels and execution times of all kernels decrease on GPU-PIM. Therefore, GPU-PIM is an appropriate configuration for all its kernels. Third, `LUH` and `FDTD` have kernels that demonstrate different behavior. Although both applications as a whole have higher execution times on GPU-PIM, some of their kernels actually benefit from GPU-PIM. For example, in `LUH` and `FDTD`, Kernel-K1 has lower execution time on GPU-PIM compared to GPU-PIC.

Thus, a careful *kernel-level* offloading strategy can perform even better than the application offloading strategy we evaluated (Section 2.3.1). Figure 2.6 illustrates the potential of kernel offloading for `FDTD`. Scenario-I and Scenario-II are the two possible application offloading strategies adopted to execute the entire `FDTD` on GPU-PIM or GPU-PIC, respectively. In the kernel offloading strategy (Scenario-III), each *kernel* of `FDTD` is offloaded to the computation engine where its execution time is lower (i.e., each kernel is offloaded to the unit it has *affinity* towards). Therefore, Kernel-K1 is offloaded to GPU-PIM and the other two kernels are offloaded to GPU-PIC. Kernel offloading saves many execution cycles (Ⓐ) even over the best application offloading strategy. However, the key challenge of kernel offloading is in identifying the *affinity* of each kernel, which we provide a new solution for in this work (Section 2.4).

Limitation II: Lack of Concurrent Utilization of GPU-PIM and GPU-PIC. An application offloading mechanism loses out on the opportunity of using *both* the GPU-PIM and GPU-PIC at the same time, because it is too coarse-grained. In contrast, if offloading is performed at the kernel level, as described before, both GPU-PIM and GPU-PIC can be utilized by *concurrently* executing *independent* kernels on them. Figure 2.6 illustrates an example of this in Scenarios IV and V. In `FDTD`, Kernel-K3 can start executing only when both Kernel-K1 and Kernel-K2 finish their executions. However, Kernel-K1 and Kernel-K2

can execute in parallel. Because of the concurrent execution of kernels, overall application execution time is reduced (B) over the best application-offloading scenario, in scenario-IV. Scenario-V in Figure 2.6 demonstrates that kernel affinity, i.e., scheduling each kernel on the execution engine that is best for the kernel's performance matters: Kernels K1 and K2 respectively have affinity for GPU-PIM and GPU-PIC, and executing them concurrently on these engines leads to even higher overall application execution time savings (C) than in Scenario-IV where the same kernels are scheduled onto the opposite engines.

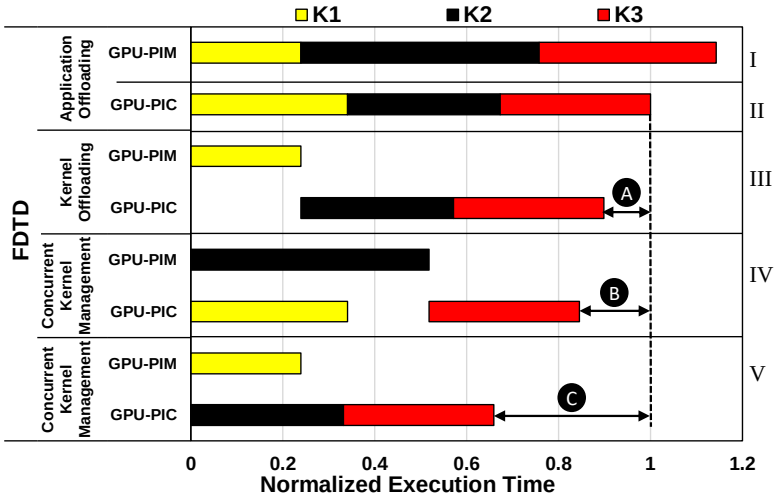


Figure 2.6: Performance advantages of kernel offloading (III) and concurrent kernel management (IV and V) mechanisms using the `FDTD` application as an example.

Our Goal. As we illustrated, while an application-level offloading strategy can improve performance and energy efficiency, a finer, kernel-level offloading strategy can provide more opportunity. Therefore, our goal is to develop mechanisms for (1) automatically identifying the architectural affinity (GPU-PIM or GPU-PIC) of each kernel in an application, and (2) scheduling kernels that can concurrently execute on different parts of the PIM-assisted GPU architecture (GPU-PIM or GPU-PIC), while balancing the execution times across architectures and maintaining a kernel's architecture affinity as much as possible.

2.4 Kernel Offloading Mechanism

This section presents an architecture affinity prediction model to enable a runtime mechanism for kernel offloading in PIM-Assisted GPU architectures.

Need for a Prediction Model. If *all* the kernels of an application prefer the same compute engine (either GPU-PIM or GPU-PIC), kernel offloading becomes equivalent to application offloading. However, if different kernels have different affinities, kernel-level offloading can yield higher performance than application offloading (Section 2.3.2). Because the CPU offloads computation to the GPU at the kernel granularity, maintaining the same granularity for PIM execution enables low-overhead computation offloading from the CPU to either GPU-PIC and GPU-PIM. To avoid the overhead of first sampling the performance of each kernel on both platforms and then deciding the appropriate platform for execution, we would like to *predict* the affinity of the kernel *before* it starts execution. To this end, we make use of a *regression model* that is composed of *predictive variables*.

Table 2.1: Metrics used to predict compute engine affinity and GPU-PIC and GPU-PIM execution time.

Primary Category	Predictive Metric	Static/Dynamic
I: Memory intensity of Kernel	Memory to Compute Ratio	Static
	Number of Compute Inst.	Static
	Number of Memory Inst.	Static
II: Available Parallelism in the Kernel	Number of CTAs	Dynamic
	Total Number of Threads	Dynamic
	Number of Thread Inst.	Dynamic
III: Shared Memory Intensity of Kernel	Total Number of Shared Memory Inst.	Static

Metrics. To build our regression model, we need to identify appropriate metrics (i.e., predictive variables) that can characterize the kernel affinity to GPU-PIC or GPU-PIM. We classify kernel characteristics into three primary categories: 1) memory intensity, 2) parallelism, and 3) shared memory (i.e., scratchpad) intensity. Table 2.1 lists these

categories along with their predictive metrics/variables.

To measure the effect of *kernel memory intensity* on performance, we consider the memory-to-compute-ratio of the instruction mix executed by that particular kernel. Memory-to-compute ratio gives insight into the level of performance the kernel can achieve with higher memory bandwidth, as the computation requires data from the memory, indicating whether the higher bandwidth available on GPU-PIM is beneficial for performance. We use the *number of cooperative thread arrays (CTAs)* (also called work groups or thread blocks) as a measure of the parallelism in the kernel. This allows us to take into account the difference in the peak instruction throughput between GPU-PIC and GPU-PIM for each kernel. For a kernel with a high number of CTAs, GPU-PIC's performance might be higher than that of GPU-PIM due to the higher number of cores in GPU-PIC. To approximate a kernel's *shared memory intensity*, we measure the total number of shared memory instructions in the kernel. A shared-memory-intensive application might not require high DRAM bandwidth, making it more suitable for GPU-PIC. Predictive metrics/variables (listed in Table 2.1 middle column) are used to help build a robust model by complementing the primary metrics of the kernel. For example, in applications such as `RED, CONV, STRM`, the number of CTAs along with the number of threads provides a notion about the CTA sizes. Usually a larger CTA has higher resource requirements, which might lead to fewer of such CTAs to be scheduled onto an SM at any given time, even in the presence of available SM resources (but *not enough resources* to schedule another complete CTA). Therefore, having more SMs, as in GPU-PIC, might lead to better performance for such a large CTA. We classify the metrics as static or dynamic, as shown in Table 2.1. Static metrics are obtained by simply parsing the source code, while dynamic metrics are input-set-based and can be known only at/after kernel launch or during runtime.

Why These Metrics? We choose only the most influential metrics to build our regression model, i.e., those metrics that contribute the most to the model's accuracy. We

define accuracy for our regression model for a kernel as either 100%, if the model predicts the kernel's affinity correctly, or 0%, if it predicts incorrectly. We experimented with building the regression model using the number of thread instructions as the only metric, and found that this model has an accuracy of 79% on the *training kernels*. Using all the metrics shown in Table 2.1 leads to an overall accuracy of 87%. The use of other characterization metrics described in Goswami et al. [90] does not further improve accuracy.

Table 2.2 provides the detailed characteristics of various application kernels, obtained via offline profiling. It also shows the primary category metrics (memory-to-compute-ratio, the number of CTAs, and shared memory intensity) and architecture affinity of each kernel (GPU-PIM (Y) or GPU-PIC (N)). The affinity of a kernel can be reasoned for most of the kernels by understanding the level of each metric category. For example, kernels such as `convolutionRows` and `prescan` have high shared memory intensity, and prefer to run on GPU-PIC because of the lower DRAM bandwidth demand and the higher instruction throughput available in GPU-PIC compared to GPU-PIM. Kernels of `GRAM` prefer GPU-PIM because of their lower parallelism and high memory intensity. Kernels such as `fdtd_step1_kernel`, which have high memory intensity coupled with high parallelism, prefer GPU-PIM, as they benefit from the higher available memory bandwidth.

Regression Model for Affinity Prediction. We build a logistic regression model [98, 99], shown in Equation 2.1, which provides a prediction for the affinity of a given kernel. A logistic regression model is a classifier and it generates a discrete output $\sigma(t)$: **1 for GPU-PIM** and **0 for GPU-PIC**. The model uses the metrics in Table 2.1 as inputs.

$$\sigma(t) = \frac{e^t}{e^t + 1} \tag{2.1}$$

where:

$\sigma(t)$ = model output ($\sigma(t) < 0.5 => 0$, $\sigma(t) \geq 0.5 => 1$)

Table 2.2: Kernel characteristics, classification, and architecture affinity. Legend: (I) Memory Intensity (Memory to Compute Ratio $= L :\leq 0.2, 0.2 < M \leq 0.3, H :> 0.3$), (II) Parallelism (No. of CTAs $= L :\leq 64, 64 < M \leq 1024, H :> 1024$), (III) Shared Memory Intensity (Total no. of Shared Mem. Inst. $= L :\leq 2.5 \times 10^5, H :> 2.5 \times 10^5$). (B) Architecture affinity (Y: GPU-PIM, N: GPU-PIC), (C) Major reasons for architecture affinity, (D) Affinity prediction by our regression model in Section 2.4) (Y: GPU-PIM, N: GPU-PIC). Only kernels that dominate application execution time are shown.

Workload	Kernel Name	I	II	III	B	C	D	Workload	Kernel Name	I	II	III	B	C	D
BFS [91]	initialize	M	H	L	Y	Cache/BW	Y	RED [87]	reduce	H	M	L	Y	BW	N
	drelax	H	H	L	N	Cache	N	SCP [92]	scalarProdGPU	L	M	L	Y	—	N
BICG [93]	bicg_kernel1	H	L	L	Y	BW	Y	SLA [92]	prescan	L	H	H	N	S.Mem.	N
	bicg_kernel2	M	L	L	N	Cache	Y	STRM [94]	kernel_compute_cost	H	M	L	Y	BW	Y
BLK [92]	BlackScholesGPU	L	M	L	Y	Cache	Y		sgemmNN_MinPlus	L	L	H	Y	Lat.	Y
	MapperCount	H	M	L	Y	BW	Y	APSP [95]	matrixMul	H	M	L	Y	Lat.	N
CCL [96]	unitBitonicSortKernel	L	H	M	N	Compute	N		apsp_seq	M	L	L	Y	Lat.	Y
	prescan	M	M	H	N	S.Mem.	Y		cuda_compute_step_factor	M	M	L	Y	BW	Y
CONV [93]	convolution3D_kernel	M	M	L	N	Cache	Y	CFD [94]	cuda_compute_flux	L	M	L	N	Cache	Y
	std_kernel	M	L	L	Y	BW	Y		cuda_time_step	H	L	L	Y	BW	Y
CORR [93]	reduce_kernel	M	H	L	N	Compute	N	CONS [92]	convolutionRows	H	H	H	N	S.Mem.	N
	corr_kernel	L	L	L	N	Cache	Y		convolutionColumns	H	H	H	N	S.Mem.	N
	fdtd_step1_kernel	H	H	L	Y	BW	Y	FWT [92]	fwtBatch1Kernel	H	H	L	Y	BW	Y
FDTD [93]	fdtd_step2_kernel	M	H	L	N	Cache	N	GUPS	RandomAccessUpdate	L	M	L	Y	BW	Y
	fdtd_step3_kernel	H	H	L	N	Cache	N	LIB [92]	Pathcalc_Portfolio	L	L	L	N	Cache	N
	gramschmidt_kernel1	M	L	L	Y	Lat.	Y		Pathcalc_Portfolio2	L	L	L	Y	BW	Y
GRAM [93]	gramschmidt_kernel2	H	L	L	Y	Lat.	Y		dfindelemin	H	H	L	Y	Cache	Y
	gramschmidt_kernel3	M	L	L	Y	Lat.	Y	MST [91]	dfindcompmin	H	H	L	N	Cache	N
	IntegrateStress	L	M	L	Y	BW/Lat.	Y		init	H	H	L	Y	Cache	Y
LUH [97]	CalcHourglassControl	H	M	H	N	S.Mem.	N	PVC [96]	MapperCount	H	H	L	Y	BW	N
	CalcFBHourglassForce	H	M	H	N	S.Mem.	N		prescan	L	H	H	N	S.Mem.	N
MVT [93]	mvt_kernel1	M	L	L	N	Cache	Y	SP [91]	dinit	H	H	L	Y	Cache	Y
	mvt_kernel2	H	L	L	Y	BW	Y		dupdateeta	H	H	L	Y	Cache	N
PR [96]	MapperCount	H	H	L	Y	BW	Y	TRA [92]	transpose_naive	M	H	L	Y	Cache	Y
	prescan	L	H	H	N	S.Mem.	N		transpose	L	H	L	Y	Cache	N

$$t = \alpha_0 + \alpha_1 x_1 + \alpha_2 x_2 + \alpha_3 x_3 + \alpha_4 x_4 + \alpha_5 x_5 + \alpha_6 x_6 + \alpha_7 x_7$$

α_i = Coefficients of the Regression Model

x_i = Predictive Metrics/Variables (Table 2.1)

To train the logistic regression model, we randomly sample 60% (15) of the 25 GPGPU applications considered in this work. These 15 applications consist of 82 unique kernels that are used as inputs for the *training* of the model. The remaining 40% (10) of the applications, consisting of 42 unique kernels, are part of the *testing* set and are used to validate the model. We perform offline profiling of entire application execution only once to train the regression model and use this built model at runtime for affinity prediction. The model is able to accurately predict the architecture affinity (either GPU-PIC or GPU-PIM)

of 83% of the test kernels. Table 2.2 shows the affinity prediction (in column D) of our model for each kernel along with the true affinity of the kernel (in column B). The major sources of inaccuracy are due to cache effects and branch divergence, which our regression model fails to accurately capture because of their heavily runtime-dependent characteristics. For example, due to the considerable amount of cache hits in kernels such as `drelax (BFS)` and `dfindelemin (MST)`, GPU-PIC outperforms GPU-PIM, even though these kernels have high parallelism and memory intensity. Yet, our model estimates incorrectly that they are better executed on GPU-PIM. We further discuss the effects of random sampling and application input on our model's accuracy in Section 2.8.

Kernel affinity could also be predicted statically, without using the dynamic metrics in Table 2.1. Using only the static metrics, the accuracy of the affinity prediction model decreases from 83% to 74%. We find that the inclusion of dynamic (i.e., input-based) metrics in the model is necessary for accurate prediction in applications such as CFD, STRM and PVC. This is because the affinity of these applications' kernels is highly influenced by input-based kernel dimensions (i.e., number of threads, CTA size) as they significantly affect the compute/resource requirements of these kernels.

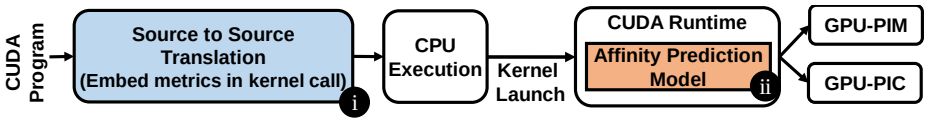


Figure 2.7: Modified CUDA runtime for kernel offloading

Implementation. Figure 2.7 shows our framework to enable kernel offloading to PIM engines. Before runtime, a simple source to source translation is performed ❶. The purpose is to compute the values of the static metrics, such as the number of memory/compute/shared-memory instructions (in terms of PTX instructions), and embed them as arguments to the kernel launch call. We extend the CUDA runtime to implement the architecture affinity prediction model ❷. The prediction model is trained *offline* and does not incur any overhead of training during online prediction of affinity. At runtime,

during kernel launch from the CPU, the dynamic metrics required for the prediction model such as the number of CTAs and the number of threads get populated with their values in the kernel launch call. Using both the static and dynamic metrics, the CUDA runtime on the host-side computes[6] the architecture affinity of the kernel using the affinity prediction model, and offloads the kernel to the architecture that is expected to provide the highest performance. Doing so avoids the overhead of kernel migration, which might arise if the kernel were to be offloaded *after* it starts execution on a less preferred architecture.

2.5 Concurrent Kernel Management

This section presents a new runtime kernel management scheme to efficiently execute multiple kernels concurrently in the heterogeneous computational units in a PIM-assisted GPU architecture.

2.5.1 Analysis

Offloading kernels appropriately to their preferred compute engine (Section 2.4) leads to performance benefit. However, due to the sequential execution of the kernels, such PIM-assisted GPU architectures are under-utilized as only either the GPU-PIC or the GPU-PIM executes a kernel at a given time, but not both. If there are independent kernels, their concurrent execution on GPU-PIC and GPU-PIM can improve overall system utilization and performance (Section 2.3.2). We develop a mechanism to achieve such concurrent execution of kernels on both GPU-PIC and GPU-PIM. To efficiently schedule kernels for concurrent execution, we need three key pieces of information: (1) kernel-level dependence information, to identify independent kernels; (2) affinity of each kernel and (3) execution time prediction for each kernel, both to decide what compute engine is the best to execute

[6]The regression model parameters are kept in CPU memory. During the API call, model-based affinity is predicted by the CPU, which requires only 15 32-bit floating point operations (7 multiplications, 7 additions and 1 comparison).

the kernel on. We first describe how this information is gathered in our runtime system.

(I) Kernel-level Dependence Information. A kernel-level data dependence graph is required to decide which kernels can execute in parallel. For our evaluations, we obtain the dependence graph of an application *for a given input* by profiling the application's execution to determine the correct and complete set of read-after-write (RAW) dependencies across the kernels. Such cross-kernel dependencies can be easily found and marked by a compiler. Most compilers targeting array-based applications already run dependence analysis for each loop nest (kernel). One could extend such an analysis to check dependence's *across* loop nests (kernels) as well.[7]

Note that, our concurrent kernel management mechanism can be directly used for applications that inherently possess concurrent kernels (which could be conveyed by the programmer), without any need for data dependence analysis.

(II) Architecture Affinity Information. We observed in Section 2.3.2 that kernel affinity information can help improve performance of concurrent kernel execution (Scenario-V, Figure 2.6). To predict the affinity of a kernel, we use the logistic regression model described in Section 2.4. This model is used to fill the GPU-PIM and GPU-PIC queues with kernels based on their affinity.

(III) Execution Time Information. Kernel dependence and affinity information is necessary, but not sufficient to balance kernel execution times across GPU-PIC and GPU-PIM. For instance, consider an application consisting of two independent kernels having affinity towards GPU-PIM. If only the affinity information is used, both these kernels are offloaded to GPU-PIM, which leads to the under-utilization of GPU-PIC (and perhaps a lost opportunity to improve performance). We address such situations by executing kernels in compute engines that do *not* satisfy the kernel affinity, if doing so would reduce overall execution time. Therefore, in this example, we might offload the kernel that has the lower

[7]Note that not all applications have multiple independent kernels and thus can take advantage of concurrent kernel execution. We observe this in `SCP` and `GUPS`.

execution time on GPU-PIC, to GPU-PIC. However, this requires the estimation of the kernel execution times on both GPU-PIC and GPU-PIM, for which we develop a model next.

2.5.2 Execution Time Prediction Model

For predicting a kernel's execution time, we build a linear regression model [100] that uses the same predictive metrics used in our architecture affinity prediction model (Table 2.1). Equation 2.2 shows the linear regression model.

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_4 + \beta_5 x_5 + \beta_6 x_6 + \beta_7 x_7 \tag{2.2}$$

where:

y = model output (predicted execution time; classified into bins using Table 2.3)
β_i = Coefficients of the Regression Model
x_i = Predictive Metrics/Variables (Table 2.1)

The model is trained using the kernel execution time information obtained from profiling the execution times of applications from the training set that were used in Section 2.4 to create the affinity prediction model. Earlier work (Dubach et al. [101]) utilizes compile-time parameters to predict the performance of an application, but exactly predicting the absolute execution time of a kernel accurately is a difficult task and exact prediction has significant error. Therefore, rather than utilizing the predicted execution time directly in an absolute manner, we classify the predicted execution time (y) into five different bins, as shown in Table 2.3. The ranges of the bins were carefully chosen by analyzing the profiled data. With such classification, we can efficiently schedule the kernels on GPU-PIC and GPU-PIM, without having to accurately predict the absolute execution times, albeit perhaps with lower benefit than if we had the correct absolute execution times.

Table 2.3: Classification of predicted execution time into bins.

Classification Bins	Very Low (1)	Low (2)	Medium (3)	High (4)	Very High (5)
Range (in Cycles)	<10K	10K-500K	500K-5M	5M-50M	>50M

Figure 2.8 shows the classification error of our regression-based execution time prediction models for GPU-PIC and GPU-PIM. Error is measured by calculating the distance of the predicted bin from the true bin normalized to the total number of bins. For example, a predicted bin of *Low (2)* with a true bin of *Very Low (1)* has an error of $(2-1)/5 = 0.2$, since the total number of bins is 5 and the distance of the predicted bin from the true bin is 1. The results show that the execution time prediction model provides a classification accuracy of 77% and 80% on the test set for GPU-PIC and GPU-PIM, respectively. The inaccuracies are mainly due to the heavily-runtime-dependent cache and branch divergence effects (as discussed in Section 2.4).

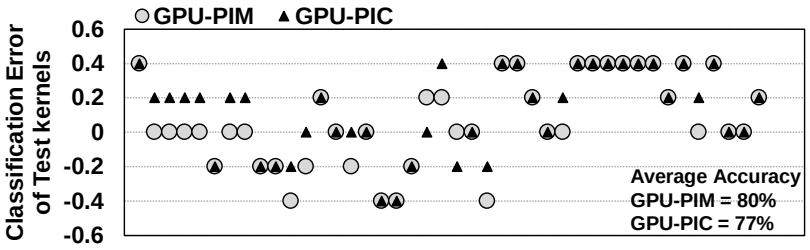


Figure 2.8: Classification error of test kernel execution times.

2.5.3 Algorithmic Details and Implementation

The main incentive to concurrently execute kernels is to maximize system utilization. For this purpose, the execution time on GPU-PIC and GPU-PIM needs to be balanced. However, the problem of balancing execution times across two architectures is equivalent to a known NP-Complete partitioning problem [102]. The partitioning problem is the task of deciding whether a given set of positive integers can be partitioned into two subsets such that the sum of the two subsets are equal. The only difference in our case is that we

have a list of pairs (instead of a single number) of positive execution times, where each pair is a tuple of <execution time on GPU-PIC, execution time on GPU-PIM>. We adopt a greedy approach to solve this problem in three steps.

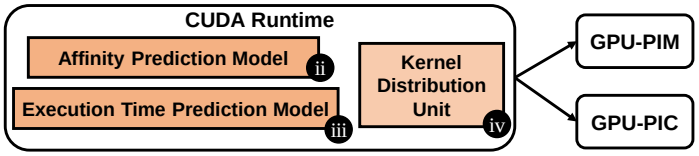


Figure 2.9: Modified CUDA runtime for concurrent kernel management.

Figure 2.9 shows the schematic of our concurrent kernel management framework. The CUDA runtime predicts the architecture affinity (ii) of all kernels launched by the CPU. The kernels are fed into their respective queues in the kernel distribution unit (iv) and the execution begins. During runtime, there might be a case where all the kernels prefer a single type of compute engine or either the GPU-PIC or GPU-PIM queue is empty but there are still kernels waiting in the other compute engine's queue. To address this, the runtime steals a waiting independent kernel from the non-empty queue to the empty queue based on the kernel's execution time prediction (iii) on the less preferred architecture. The runtime prefers to steal the *first* independent kernel that has a predicted execution time (on the less preferred architecture) smaller than the remaining execution time of the currently executing kernel. Algorithm 1 describes this greedy process of kernel stealing that enables concurrent kernel management.

2.6 Evaluation Methodology

Infrastructure. We modified the cycle-accurate GPGPU-Sim v3.2.2 [103, 104] to simulate our PIM-Assisted GPU architecture. To enable the execution of two different GPUs, we created two clusters of SMs, one for GPU-PIC and another for GPU-PIM. GPU-PIC and GPU-PIM do *not* concurrently work on the same data. Therefore, to maintain coherence between them, we flush the L2 cache in GPU-PIC after kernel

Algorithm 1 Runtime Queue Management

▷ $time(kernel, engine)$ returns the time-bin of the kernel if it is executed on the given engine.
▷ Let X and Y represent the two different engines (GPU-PIC and GPU-PIM).
▷ $independent\_kernels(queue)$ returns the list of the kernels that can run concurrently with the kernels currently executing and have no past dependencies.
▷ $time\_executed(kernel)$ returns the amount of time the kernel has been executing.

```
if X = Idle && X.queue = ∅ && Y.queue ≠ ∅ then
    if independent_kernels(Y.queue) ≠ ∅ then
        for each kernel in independent_kernels(Y.queue) do
            if time(kernel, X) ≤ (time(kernel_running,y, Y)
                                  −  time_executed(kernel_running,y)
                                  +  time(kernel, Y) ) then
                Execute "kernel" on X
                Break
            end if
        end for
    end if
end if
```

▷ Similarly, repeat the process for Y

execution. As the L1 caches are write-through, they do not need to be flushed. There is no explicit synchronization needed between GPU-PIC and GPU-PIM as they never share data during concurrent execution. The kernel distribution unit in the CUDA runtime, shown in Figure 2.9, checks for any inter-kernel dependencies to avoid concurrent scheduling of dependent kernels.

For concurrent kernel execution, we use CUDA Streams, which are supported by GPGPU-Sim. A stream is a series of kernel launches and memory operations between the CPU and GPU that are executed sequentially. Different streams are capable of executing concurrently depending on available resources. In our simulation framework, we create two CUDA Streams: `PIC_CUDAStream` and `PIM_CUDAStream`. GPU-PIM and GPU-PIC are assigned their own streams, thereby making the execution of the streams concurrent. We modified the CUDA runtime support in GPGPU-Sim to overload the API calls with the kernel metrics and added support for the *architecture affinity prediction model* (Section 2.4), *execution time prediction model* (Section 2.5.2), and *kernel distribution unit* (Section 2.5.3). During application execution, our runtime framework is able to read the

Table 2.4: Parameters of the simulated system.

GPU-PIC Features	1.4GHz, 32 cores, 32 SIMT width, GTO warp scheduler [77]
Shared L2 (GPU-PIC)	16-way 64KB/memory channel, 128B cache block size
GPU-PIM Features	1.4GHz, 8 cores, 32 SIMT width, GTO warp scheduler [77]
Resources per Core [105–107]	16KB shared memory, 16KB register file, Max. 1536 threads (48 warps, 32 threads/warp)
Private Caches per Core [105–107]	16KB L1 D-cache, 12KB T-cache, 8KB C-cache, 2KB I-cache, 128B block size
Memory Model	12 Memory Controllers, FR-FCFS, 8 banks/MC, 924 MHz Partition chunk size: 256 bytes [108] t_{CL} = 11, t_{RP} = 11, t_{RC} = 39, t_{RAS} = 28, t_{CCD} = 2 t_{RCD} = 11, t_{RRD} = 5, t_{CDLR} = 5, t_{WR} = 12
Bandwidth	88.8GB/s (Interposer link per GPU-PIM stack) 355.2GB/s (within GPU-PIM stack)
Interconnect (GPU-PIC) [107]	1 crossbar/direction (32 cores, 12 MCs), flit size=32B, 1.4GHz, islip VC & switch allocators
Interconnect (GPU-PIM) [107]	1 crossbar/direction (8 cores, 12 MCs), flit size=32B, 1.4GHz, islip VC & switch allocators

kernel metrics passed along with the kernel launch call, obtain the runtime metrics, and predict the affinity of the kernel.

Workloads. We chose CUDA applications from various benchmark suites (NVIDIA SDK [92], Rodinia [94], Shoc [87], Polybench [93], Mars [96], LonestarGPU [91]) and several other applications (e.g., LUH [97], APSP [95], GUPS). We collect the results at kernel boundaries to ensure that all comparisons are for the same amount of work completed across different executions. We only simulate the portions of code that are executed on GPU.

Simulated Systems. Table 2.4 provides the details of the simulated GPU-PIC and GPU-PIM configurations. We assume that GPU-PIC has 32 GPU cores and GPU-PIM has 8 GPU cores underneath a 3D memory stack. We compare this design to a conventional baseline GPU architecture that has 40 GPU cores. These two configurations have the same peak execution throughput. The L2 cache size of the baseline GPU is equal to that of GPU-PIC (768 kB). GPU-PIM does not have an L2 cache Section 2.2.2). All our simulations

on the baseline GPU maintain (if any) inter-kernel data reuse, i.e., the L2 cache is *not* flushed in-between the execution of two kernels, unlike concurrent GPU-PIC and GPU-PIM execution on our proposed architecture.[8]

To simulate the timing of 3D-stacked DRAM, we use the timing parameters provided by Jevdjic et al. [109]. We use GPUWattch [88] for power analysis of GPU-PIC and GPU-PIM cores, caches, and interconnect. For DRAM energy analysis, we augment this model with a simple linear equation adding the wire transfer energy numbers given by Keckler et al. [39] to the DRAM read/write energy for a Hybrid Memory Cube DRAM [73]. We faithfully model the bandwidth, latency, and timing of 3D-stacked DRAM. GPU-PIM has access to the full bandwidth of 3D-stacked DRAM, whereas GPU-PIC has limited bandwidth as it is pin-limited. Table 2.5 provides the parameters we use for calculating DRAM energy. We assume a crossbar interconnect between the private L1 caches and the shared L2 cache, keeping it consistent with the currently available GPUs [35].

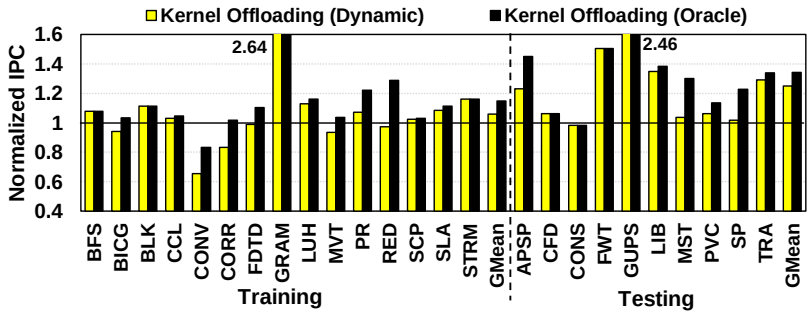
Table 2.5: Parameters of our DRAM energy model.

Energy per bit [73]	13.7 pJ/bit
Wire Energy (256 bits, 10 mm) [39]	310 pJ
Assumed distance between GPU-PIC and GPU-PIM	20 mm

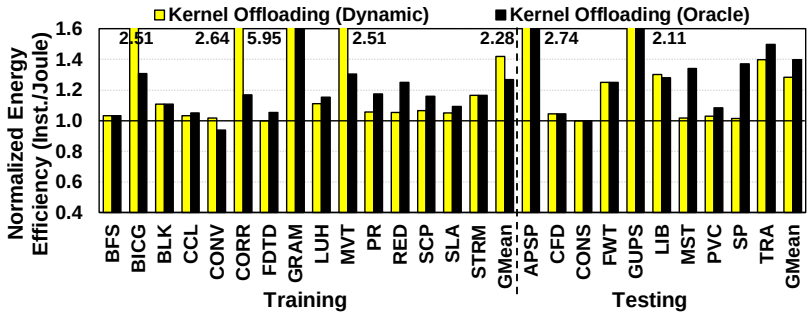
2.7 Experimental Results

We compare the performance and energy efficiency of our kernel offloading and concurrent kernel management schemes with their oracle counterparts. The oracle schemes make ideal

[8]We also considered using a baseline architecture capable of concurrent kernel execution on a *partitioned* set of 32 SMs and 8 SMs, but there are two issues with such a baseline. First, GPGPU applications tend to be highly parallel and fill the entire set of SMs provided, and we take this into account in our 40-SM baseline. Second, we still need a mechanism to decide which partition to launch each of the kernels. Depending on its characteristics, a kernel might prefer lower or higher number of SMs. Our new kernel offloading mechanism may be modified to make such a baseline work, but we leave this as future work. For these reasons, we normalize all results to a baseline with 40 SMs and no concurrent kernel execution.



(a) Performance.



(b) Energy Efficiency.

Figure 2.10: Impact of our Kernel Offloading scheme.

offloading and concurrent execution decisions by profiling each kernel with its runtime input and obtaining completely accurate execution times for each kernel on GPU-PIC and GPU-PIM (instead of using regression models). Therefore, the oracle schemes provide the best possible performance for each application. The oracle kernel offloading scheme finds the correct kernel affinity for each kernel and schedules it for execution on the best engine. The oracle concurrent kernel management scheme provides the minimal execution time for an application by utilizing both GPU-PIC and GPU-PIM while respecting kernel dependencies. We *normalize* the performance and efficiency results to those of the 40-SM baseline GPU described in Section 2.6. We report the results separately for the applications used for training and testing, to show the effectiveness of our model.

Effects of Kernel Offloading. Figures 2.10a and 2.10b show the performance and energy efficiency benefits of our dynamic kernel offloading scheme, respectively. Our

technique increases the performance and energy efficiency of the testing set applications by 25% and 28%, respectively. The oracle kernel offloading scheme provides 34% and 40% average performance and energy efficiency improvement, respectively, on the testing set. The inability of our scheme to perform as good as the oracle scheme is due to the mispredictions of kernel affinity by our regression model.[9]

The inaccuracies of our prediction model (Section 2.4) usually cause some kernels (e.g., of `CONV` and `CORR`) to be incorrectly offloaded to GPU-PIM, leading to performance losses. However, because GPU-PIM is significantly more energy efficient than GPU-PIC, in most of the mispredicted kernels (e.g., of `APSP` and `LIB`; see Figure 2.10b), the overall energy efficiency of our scheme is still higher than that of the oracle's, which is optimized for performance, not energy.

Figure 2.11 shows the percentage of execution time when GPU-PIC and GPU-PIM are executing kernels. Applications such as `LUH`, `PR`, and `SLA` have kernels that exhibit different architecture affinities within the application and thus benefit from kernel offloading over application offloading. Our model is able to capture this variation in affinity and we utilize the opportunity that kernel-level offloading presents us as mentioned in Section 2.4. In this scheme, GPU-PIC and GPU-PIM *do not execute concurrently*, leading to under-utilization of the system.

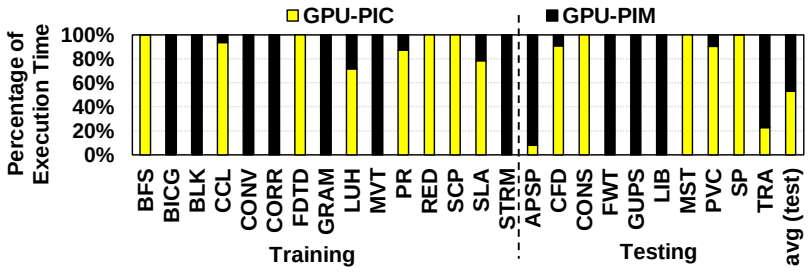


Figure 2.11: Percentage of execution time GPU-PIM and GPU-PIC execute kernels with our kernel offloading scheme.

[9]In some cases where computation parallelism is the main bottleneck, e.g., for `CONV`, even the oracle scheme loses performance compared to the baseline because the baseline can utilize all the 40 GPU cores for each kernel whereas the PIM-assisted GPU architecture can utilize either 32 or 8.

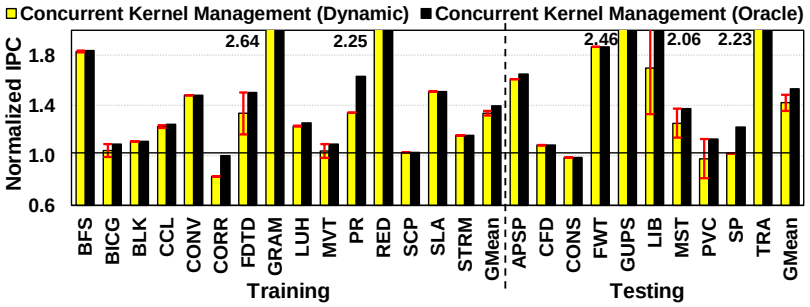
We make three observations based on these results. First, kernel offloading gives the same results as the best application offloading (as we showed in Figure 2.4) for applications such as BFS, BLK, SCP, and GUPS. This is because these applications either consist of a single kernel (BLK, SCP and GUPS), or all the kernels of the application prefer the same engine (BFS). Second, kernel offloading performs better than application offloading in applications such as FDTD, PR, and PVC. This is due to the benefits of fine-grained offloading, discussed in Section 2.4, which selects the fastest execution engine for each kernel. Third, our scheme performs worse than application offloading for applications such as BICG, CORR, and MST, due to mispredictions in kernel affinity. BICG has two kernels, one of which prefers GPU-PIC and the other GPU-PIM. Due to mispredictions, the kernel that prefers execution on GPU-PIC is executed on GPU-PIM, leading to lower performance compared to application offloading, which offloads the entire application to GPU-PIC. These mispredictions are due to data reuse at the L2 cache, which make GPU-PIC faster than GPU-PIM (lacks an L2 cache).

Effects of Concurrent Kernel Management. Figures 2.12a and 2.12b show the performance and energy efficiency benefits of our concurrent kernel management scheme, respectively. On average, our management scheme improves performance and energy efficiency by 42±4% and 27±2%, respectively, over the baseline across the test set.[10] The oracle scheme on the test set provides 53% and 33% improvements in performance and energy efficiency, respectively. Similar to our kernel offloading scheme, affinity mispredictions cause performance penalties, but they improve energy efficiency in general.

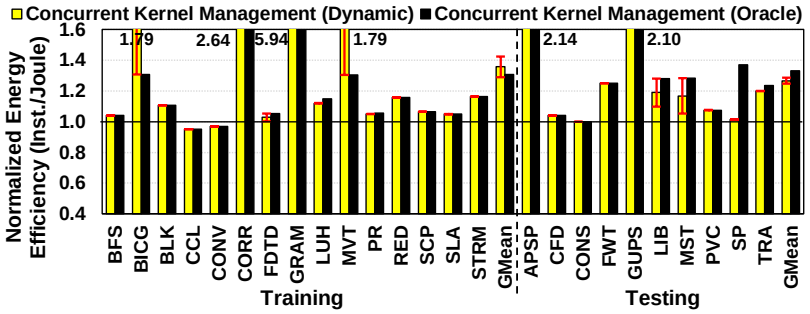
Figure 2.13 shows the percentage of execution time when both GPU-PIC and GPU-PIM are executing kernels concurrently.[11] It can be seen that the applications with high

[10] As a result of the dynamic nature of kernel scheduling and the greediness in our scheme, different concurrent kernel executions might be possible for the same application, which might result in different performance and efficiency. This arises because multiple independent kernels can be predicted to be in the *same execution time bin*. Any one of these kernels could be picked by our concurrent kernel scheduler as they are at equal "priority". The error bounds in Figure 2.12a and 2.12b show the variation in performance and energy efficiency due to different choices made by the scheduler at runtime.

[11] The breakdown for the best-case of concurrent kernel execution is shown.



(a) Performance.



(b) Energy Efficiency.

Figure 2.12: Impact of our Concurrent Kernel Management scheme.

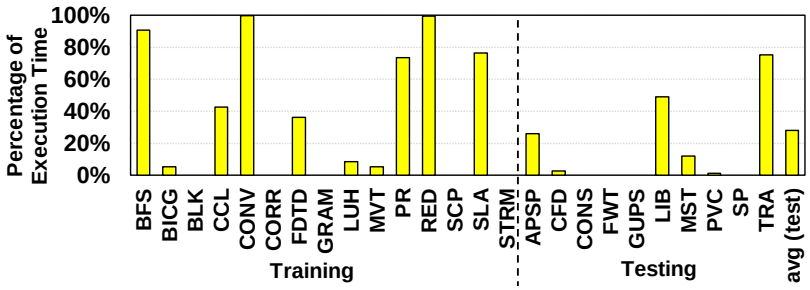


Figure 2.13: Percentage of execution time when kernels are concurrently running on GPU-PIM and GPU-PIC with our concurrent kernel management scheme.

concurrency across the engines are the ones that gain the most in terms of performance.

We make four observations based on these results. First, our scheme performs the same as application offloading in applications such as `BLK`, `GRAM`, `FWT`, and `GUPS`. There is no kernel-level concurrency in these applications (see Figure 2.13). `BLK` and `GUPS` consist of a

single kernel; GRAM and FWT have no independent kernels, leading to sequential execution of kernels in all of these applications.

Second, applications such as FDTD, PR, LIB and MST fail to perform as good as the oracle scheme because their execution times are not predicted accurately (even though their affinity predictions are correct). Our heuristic checks for the kernel's predicted runtime on the unsuitable platform, and decides whether to wait for its preferred platform to be free, or to execute it concurrently on the less-suitable yet under-utilized platform right away. Our execution time prediction fails to calculate the exact execution time bin, and leads to sub-optimal scheduling decisions in these applications. Third, applications such as BICG, MVT, PVC and SP under-perform due to the mispredictions in kernel affinity. In SP, one of the two kernels is mispredicted for GPU-PIM, and both of the kernels are executed on GPU-PIM. Following our heuristic of running the kernel with shorter execution time based on the execution time bins does not help because both kernels fall into the same bin. Therefore, the heuristic arbitrarily chooses one kernel for GPU-PIM and the other for GPU-PIC, which negatively affects performance as one of the kernels takes considerably longer on GPU-PIC even though they fall in the same execution time classification bin. Also, due to the lack of many kernels, there is no leeway for mispredictions. The scenario is similar for the other applications as well. APSP also suffers from this situation, but because this application has many (200+) kernels that execute, the mispredicted kernels do not dominate the execution time. Thus, we still get significant performance improvement on APSP. Finally, for applications such as CCL, RED, CFD and FWT, we are able to achieve comparable performance to that of the oracle scheme, which is very significant. In RED, even though the affinity is incorrectly predicted, due to kernel stealing, we are able to offload the appropriate kernels to their preferred architecture, which leads to significant performance improvement.

We conclude that our kernel offloading and concurrent kernel management schemes lead to significant average performance and energy efficiency improvements across 25 applications

we evaluated for both training and testing.

2.8 Sensitivity Studies

We perform multiple sensitivity studies to understand: 1) the impact of architectural decisions for GPU-PIM, 2) sensitivity of the regression model to the testing set and different application inputs, 3) opportunities and challenges of utilizing multiple GPU-PIMs.

2.8.1 GPU-PIM Design Choices

L2 Cache and Core Count. We analyzed the effect of caches on GPU-PIM's performance. There is 9% slowdown when a 384kB L2 cache is added to GPU-PIM, due to the additional latency the L2 cache introduces to the memory request path, which is heavily used in the presence of memory-intensive kernels. We also varied the number of SMs in GPU-PIM from 4 to 16. With 8 SMs, GPU-PIM achieves within 30% of the performance of 16 SMs. Importantly, 8 SMs is thermally feasible (<50W), but more SMs likely reduce the feasibility of GPU-PIM.

2.8.2 Regression Model

Training Set. To evaluate our mechanisms' sensitivity to applications used in the training set, we selected a completely different training set (by picking applications to be included in the training set in a random manner) and rebuilt the regression model. We found the accuracy in predicting the architecture affinity of the test kernels to be 81%. We also performed a semi-random sampling, where 20% of the *most influential* applications (that affect the accuracy of the model) were made a part of the training set and the remaining 40% were chosen randomly to be included in the training set. This leads to an accuracy

of 88% for the test kernels. To study the impact of varying the size of the training set on the regression model, we build models using three different training set sizes: 40%, 60% and 80% of the applications. The accuracy of these models on the test set were 70%, 83% and 81%, respectively. As the training set size increases, accuracy improves to a point, but further increase over-fits the model (i.e., when more than 60% of the applications in the training set), after which accuracy starts decreasing.

Sensitivity to Application Input Set. Figure 2.14 shows the performance of our kernel offloading scheme on 8 GPGPU applications with different input sets. For applications such as `CFD, STRM, PVC`, we are able to predict the affinity accurately. These applications are able to change their parallelism (number of CTAs, number of threads) depending on their inputs, enabling accurate affinity prediction. For `RED Input-2`, we capture affinity accurately, but `RED Input-1` has mispredictions. This is because Input-1 fits inside the cache and has high reuse, unlike Input-2, whose working set is larger than the cache. For applications such as `BFS, MST, SP`, we predict affinity incorrectly, which leads to suboptimal performance. These applications are heavily dependent on their input data and it is difficult for our model to perfectly account for such very irregular cases. In `MST Input-1`, different kernels prefer different affinities, whereas `MST Input-2` contains kernels that always prefer to execute on GPU-PIM.

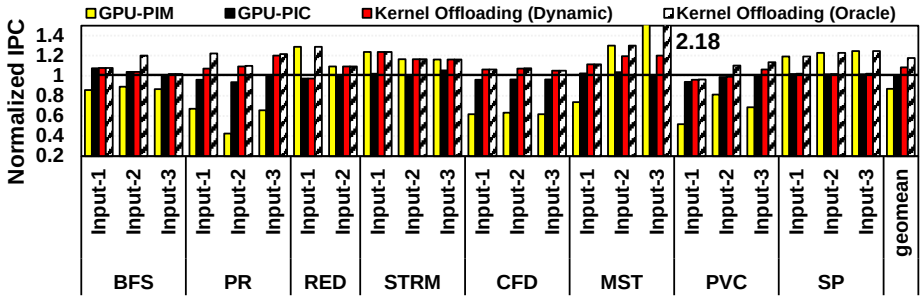


Figure 2.14: Affinity prediction model's sensitivity to input.

2.8.3 Systems with Multiple GPU-PIMs

Number of GPU-PIMs. We experimented with two other configurations that have 2 and 4 GPU-PIMs, respectively, while increasing the GPU-PIC memory bandwidth by 2× and 4× (due to multiple memory links between GPU-PIC and GPU-PIMs). Assuming perfect data placement (i.e., data needed by a GPU-PIM is located in its local stacked memory), application offloading onto only GPU-PIMs gives an improvement of 31% and 51% in performance and energy efficiency, respectively. With the *best application offloading* scheme, we see improvements of 53% and 65%, respectively. These trends are similar to the trends discussed in Section 2.3, and indicate that our schemes are likely to be scalable with number of GPU-PIMs.

Data Placement. Earlier results with multiple GPU-PIMs assumed *perfect* data placement such that each GPU-PIM has the data it needs in its local memory stack. However, this might not always be possible. Therefore, architectures comprising multiple GPU-PIMs might suffer from memory transactions that are requested and served by different (non-local) memory stacks. Moreover, these transactions have to go through the GPU-PIC since there are no direct communication channels between the different GPU-PIMs. With the default CTA scheduler and the default address mapping, we find that approximately 50% of memory transactions are non-local. We changed the striping of data from 256Bytes/bank to 32KBytes/bank, causing non-local accesses to reduce by 8-10% for applications such as `MST`, `STRM` and `LIB`. By modifying the CTA scheduler and/or address mapping intelligently, it is possible to minimize non-local accesses, as shown by a recent work [76].

2.9 Related Work

To our knowledge, this is the first work that comprehensively investigates kernel-level offloading mechanisms in PIM-Assisted GPU Architectures. It is the first to develop automated models and methods for (1) offloading kernels to PIM units, (2) concurrently executing kernels on different heterogeneous compute engines in a PIM-Assisted GPU Architecture, to maximize system performance and efficiency. We briefly discuss research in related areas.

Processing-in-Memory (PIM) Architectures. There is a substantial body of work on PIM that explores placing computation units within memory (e.g., [60, 65–70, 110–117]). 3D-stacked memory technology brings new dimensions and better feasibility to PIM-based architectures [61–63, 71, 72, 76, 118–121, 121–124]. Our work is most closely related to the concurrent work of Hsieh et al. [76], which proposes programmer-transparent schemes for offloading code segments to PIM cores and for co-locating code and data together in a 3D-memory stack. They use a compiler-based technique to find the code segments to offload to the PIM compute units based on a cost-benefit analysis. Our work does not require sophisticated compiler support as it performs scheduling at the kernel level and kernels are already well designated in modern GPU applications. Farmahini-Farahani et al. [125] propose an architecture that reduces data transfers by stacking accelerators on top of off-chip DRAM devices. In the context of GPUs, Zhang et al. [30] propose TOP-PIM, a throughput-oriented PIM-Assisted GPU architecture. They show significant energy efficiency improvements by offloading GPU applications closer to memory. However, they evaluate executing the *entire application* on either the host or GPU-PIM. Our work builds upon a similar architecture and proposes mechanisms to more efficiently utilize such an architecture by performing scheduling at the finer-grained kernel level.

Machine Learning-based Prediction Models. Machine learning techniques have been widely deployed for performance and power prediction models (e.g., [126–136]). Wu et

al. [131] propose a GPU performance and power model that uses machine learning techniques to predict the behavior of incoming applications from profiled data. Ardalani et al. [132] propose a performance prediction model that takes as input a single-threaded CPU version of an application and predicts the performance for its GPU port. Panwar et al. [133] present an online kernel characterization technique and performance model to estimate the performance of a kernel on different GPU architectures. Ipek et al. [134, 135] develop models for performance prediction of parallel applications and for aiding architectural space exploration. We use a regression-based approach to develop our affinity and execution time prediction models for PIM-Assisted GPU architectures.

Task Scheduling. There has been considerable work done in the domain of task scheduling to improve load balance and performance in both homogeneous and heterogeneous systems (e.g., [50, 137–148]). Aji et al. [149] design an OpenCL runtime called MultiCL, which can effectively map the command queues onto the best device for high performance. Their runtime scheduler involves static device profiling, dynamic kernel profiling, and dynamic device mapping. Chen et al. [150] study a task queue based dynamic load balancing mechanism for a multi-GPU setup. None of these works examine scheduling or load balancing issues in a PIM-Assisted GPU Architecture. In this work, we develop new affinity and execution time prediction models to efficiently schedule kernels to heterogeneous compute units in such an architecture.

2.10 Chapter Summary

We developed two new code scheduling techniques that enable effective use of processing-in-memory (PIM) mechanisms in PIM-Assisted GPU architectures, where a conventional GPU is augmented with 3D memory stacks that house simpler GPUs in their logic layers. First, a *kernel offloading* mechanism that accurately decides what code portions to offload to the GPUs in the 3D memory stack, using a new regression-based kernel affinity prediction

model. Second, a *concurrent kernel management* mechanism that uses the affinity prediction model, a new kernel execution time prediction model, and kernel dependency information to decide which kernels to schedule concurrently on both the main GPU cores and the GPU cores in memory stacks. These two mechanisms operate at the *kernel-level*, which simplifies scheduling and management, and makes our approach transparent to programmers and compilers, as code is offloaded from the host system to a GPU system at the kernel granularity in modern systems. We have comprehensively evaluated both of our mechanisms and shown that 1) they provide significant performance and energy efficiency improvements across a wide variety of GPU applications and 2) the improvements we obtain are robust to changes in the training set for our regression model and changes in system parameters. We conclude that our kernel-level scheduling mechanisms can be an effective runtime solution for exploiting processing-in-memory in modern GPU-based architectures.

Chapter 3

Enabling Opportunistic Computations on Throughput Processors for Reduced On-Chip Data Movement

Data transfer overhead between computing cores and memory hierarchy has been a persistent issue for von Neumann architectures and the problem has only become more challenging with the emergence of manycore systems. A conceptually powerful approach to mitigate this overhead is to bring the computation closer to data, known as Near Data Computing (NDC). Recently, NDC has been investigated in different flavors for CPU-based multicores, while the GPU domain has received little attention. Recall that, the previous chapter dealt with computation offloading between GPU and PIM units to improve performance. In this chapter, we present novel NDC solutions for GPU architectures with the objective of minimizing on-chip data transfer between the computing cores and last-level cache.

3.1 Introduction

The memory wall has been a major impediment to designing high performance von Neumann style computing systems. With the recent manycore trend, the cost of moving data from different levels of the memory hierarchy to the cores has further accentuated the memory wall problem [39, 151, 152]. This is likely to become more challenging with technology scaling as more transistors are squeezed into larger dies exacerbating the relative cost of moving data to that of a compute operation. Thus, the performance and energy overheads of data movement would be a continuing non-trivial challenge in designing high-performance, energy-efficient systems.

Processing-In Memory (PIM) and Near Data Computing (NDC) [40, 60, 61, 65, 76] concepts have been proposed to minimize these overheads by moving computation closer to data. Recent advances in technology have made computational logic cheaper, plentiful and easier to integrate thereby making NDC a feasible approach. NDC is an abstract framework that requires answering several key design questions — *what to offload*, *where to offload* and *when to offload*. For example, in a traditional multi-level memory hierarchy, computation offloading can be done to the on-chip caches or off-chip DRAM and the granularity of computation would vary depending on where the computation is done.

Traditionally, NDC mechanisms have allowed CPUs to further improve their performance and energy-efficiency. However, to our knowledge, the NDC concept has been little explored in the context of GPU architectures. As GPUs are likely to play a major role in achieving energy-efficient Exascale systems, further scaling of their performance and energy-efficiency is a critical task [3]. With technology scaling, the relatively high data movement costs in GPUs are starting to bottleneck further energy-efficiency improvements. The overheads of data movement are greatly accentuated in GPUs due to three major reasons. First, GPU applications are highly data parallel, and therefore, work on large amounts of data. Second, as GPUs scale, compute throughput improvements are

outpacing the memory bandwidth improvements, hence, worsening the memory wall problem. Finally, state-of-the-art GPUs have many more cores [25, 26] that need a larger network-on-chip [153, 154] to connect them with memory, leading to an increased data movement and traversal costs. Furthermore, many prior works [155, 156] have identified on-chip interconnect bandwidth to be a bottleneck as well. These reasons have made it imperative to find novel mechanisms to further enhance energy efficiency of GPUs. Recent works [40, 76] have studied the PIM concept in GPUs, where they minimize off-chip data movement by facilitating computation at 3D-stacked DRAMs. To the best of our knowledge, prior NDC efforts do not optimize on-chip data movement in GPUs. Thus, the goal of this work is to minimize on-chip data movement between the GPU cores and the LLC for improving performance and energy efficiency.

It is non-trivial to make GPUs amenable to NDC mechanism due to three main reasons. First, GPUs are load-store architectures, and the ALU operations are performed only on registers. Therefore, finding candidates for offloading involves searching for suitable instruction sequences. Second, GPUs are SIMT architectures and their fetch, decode and wavefront scheduling units are tuned for hiding memory latency by executing many parallel instructions. Specifically, GPUs try to execute as many instructions as possible from other threads to hide a cache miss, which in turn would lead to longer latencies to offload a set of instructions from a given thread. Hence, efficiently offloading a set of instructions involving multiple loads that incurred L1 misses from a given thread, while minimizing the execution of the other threads is a challenging issue. Third, due to the need for massive memory-level parallelism, data is interleaved across multiple memory partitions. If offloaded instructions require data from multiple partitions, it is not straightforward to find a "single" location to perform the offloaded computation.

The decision of *what to offload* is crucial for the effectiveness of NDC mechanisms. Ideally, one would like to find sequences of instructions that can be offloaded as a whole to minimize data movement. Our NDC mechanism first finds suitable instruction sequences

for computational offloading, called *offload chains*, that corresponds to commonly used basic, high level instructions that appear across many applications. We use a compiler pass to tag the offloadable chains. Next, to address *where to offload* the computation, we introduce the term, Earliest Meet Node (EMN), which is the first intersecting node of the loads in the offload chain on their traversal paths. We then form a "*ComputePacket*" of these instructions that can be pushed to the EMN for computation. To perform the offload, we propose two computational offloading mechanisms. The first one, *LLC-Compute*, is employed when the EMN is the LLC node itself. By providing the required computation hardware in the LLC, the computation can be performed there. The second scheme, *Omni-Compute*, is employed when the EMN is an intermediate GPU node in the network; it offloads the computation to another GPU node, which is en route between the source GPU node and LLCs. *Omni-Compute* provides the necessary additional bookkeeping logic to the GPU cores to be able to compute instructions offloaded by other cores beyond the logic needed to execute its own instructions. We show that simply sending all offloadable chains to the EMN for computation is not optimal in terms of performance due to its implications on data locality. Therefore, *when to offload* the chains is critical for the efficiency of the NDC mechanism.

To our knowledge, this is the first work that considers reducing on-chip data movement in GPUs by *opportunistically* offloading computations either to (1) the last-level-caches, or (2) the ALU of another GPU core that would result in the lowest on-chip data movement for that computation. This chapter makes the following major **contributions**:

- It proposes two NDC schemes to facilitate computational offloading in GPUs. It shows that basic forms of *load-compute-store instructions* are ideal candidates for offloading.
- It provides (i) compiler support for tagging offloadable instruction chains, (ii) the architectural modifications required for forming *ComputePackets*, (iii) the computational units and (iv) the controller units for LLC-Compute and Omni-Compute with negligible hardware overheads.

- It comprehensively evaluates our proposed NDC mechanisms using nine general purpose GPU workloads. The LLC-Compute technique provides, on an average, 19% and 11% improvement in performance (IPC) and performance/watt, respectively, and 29% reduction in on-chip data movement compared to the baseline GPU design. The Omni-Compute design boosts these benefits to 31%, 16% and 44%, respectively, by providing additional offloading opportunities.

3.2 Background

Programming Environment: The parallel parts of CUDA [92]/OpenCL™ [24] applications, are called "kernels". A kernel contains multiple workgroups. The GPU hardware dispatcher schedules workgroups onto cores. The threads within a workgroup are organized into groups, called "wavefronts". Wavefronts are the granularity at which the GPU schedules threads into SIMD pipeline for execution.

Architecture: The evolution of GPU architecture indicates that the GPU compute capability is scaling along with the number of cores. As stated in Chapter 1, recent AMD Radeon™ RX Vega 64 GPUs [26] and NVIDIA® Tesla® V100 GPUs [25] are already equipped with 64 compute units (CUs) and 80 streaming multiprocessors (SMs), respectively. Figure 3.1 shows our baseline GPU architecture. Each core has a private L1 data cache, a texture cache and a constant cache, along with a software-managed scratchpad memory (shared memory). The cores are connected to shared last-level cache (LLC) partitions via a Network-on-Chip (NoC). Each LLC partition is directly connected to a memory controller. In our baseline architecture, we model a GPU with 56 cores and 8 last-level cache (LLC)/memory controllers (MCs) connected through an (8×8) mesh-based NoC [29, 157] (YX routing) as shown in Figure 3.1. The MC placement in our baseline GPU is a variation of the *checkerboard* configuration, which is shown to be throughput-effective for GPUs when compared with other MC placements [153]. This

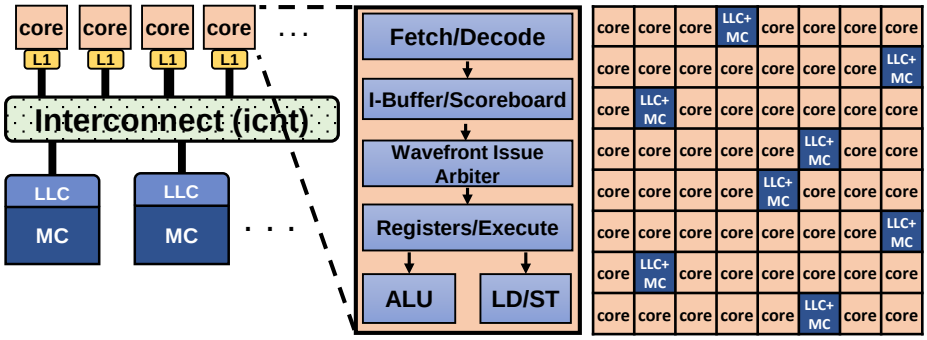


Figure 3.1: Baseline architecture.

placement allows for efficient link usage, while minimizing link hotspots that arise due to dimensional routing for traditional MC placements.

Figure 3.1 shows the micro-architecture of a core. To execute a wavefront, the fetch unit first fetches the next instruction from instruction cache based on the program counter (PC). The fetched instruction is decoded and placed into an instruction buffer for execution. This buffer is hard partitioned for each wavefront, and the decoded instructions are stored on a per-wavefront basis. The fetch-decode happens in a round-robin manner for all the wavefronts in a core. This keeps the pipeline full so that on a context switch, a new wavefront is ready to be issued immediately. In our baseline configuration, the buffer can hold two decoded instructions per wavefront, limiting only two instructions from a particular wavefront that can be issued continuously for execution. Also, when a wavefront encounters an L1 miss, the wavefront scheduler will switch out the current wavefront with another ready wavefront to hide the memory latency.

3.3 Motivation and Analysis

Today's GPUs use separate instructions for memory accesses and ALU operations. Figure 3.3 shows an example code fragment of an addition operation performed on operands `a` and `b` and stored in operand `c`. This instruction is broken down into multiple

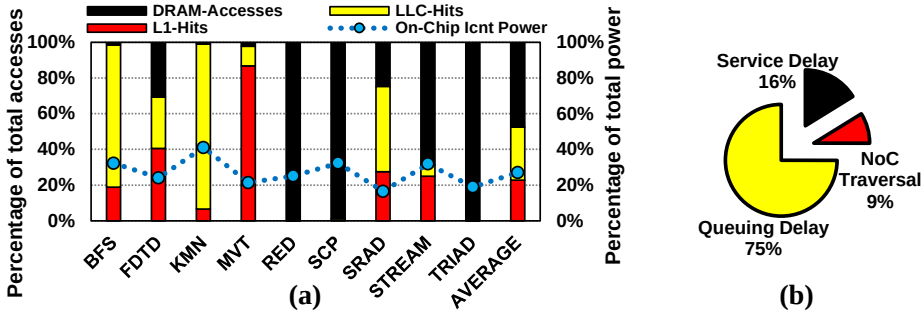


Figure 3.2: (a) Breakdown of memory requests across the memory hierarchy and the on-chip interconnect power as a percentage of the total GPU power. (b) Percentage of time spent by memory requests (L1 misses) for NoC traversal, queuing delay at the injection/ejection ports and LLC/DRAM service. The average of all applications is shown.

load/alu/store operations in a load-store architecture. The loads fetch the required data from the caches/memory into the registers of the core. The data could potentially come from the L1 cache, LLC or the off-chip main memory. Ideally, if all the memory requests are hits at the L1 cache, there will be minimal undesirable data movement apart from the data movement caused by cold misses.

Figure 3.2(a) shows the amount of data being fetched from different levels of the memory hierarchy for nine applications. On average, only 23% of the available data is fetched from the L1 cache, while 30% is fetched from the LLC and the remaining 47% is fetched from the DRAM. We provide the detailed performance and energy efficiency evaluation methodology in Section 3.5. Figure 3.2(b) shows the breakdown of the average memory latency of requests (L1 misses) across nine applications during their traversal from the cores to the LLC nodes and back. We observe that almost 75% of the memory latency is due to the queuing of the packets in the injection queue at the LLC nodes, 16% of the time is spent servicing these requests, while the rest 9% is taken up by the NoC traversal. NoC traversal constitutes the cost of VC allocation + route computation + switch allocation and link traversal [158]. These observations are mainly due to three reasons: (1) read to write ratio; (2) burstiness of the requests and; (3) traffic patterns in GPUs. Note that, the request traffic sends requests from multiple cores to a few LLCs/MCs, while in the response network, a few

LLCs/MCs send data (with a higher payload) to multiple cores. These reasons cause a disparity between the effective request/service rate of the cores (more request packets can be sent) and LLCs nodes (fewer response packets can be sent). To summarize, *77% of the requested data is transferred over the on-chip interconnect, which contributes to an average power consumption of 27% of the total GPU power.* Therefore, the implications are twofold. First, we need to reduce the amount of data transfers over the network, and second, exploit computations rather than waiting for data that is queued up in the network.

3.3.1 Analysis of Data Movement

To reduce the on-chip data movement, we need to understand and analyze the flow of data from different levels of the memory hierarchy to the cores. Let us consider two scenarios for the flow of data during the execution of code snippet on core 15 in Figure 3.3.

Scenario 1: Let both the load requests and the store request go to the same LLC partition (assume in Figure 3.3, the memory requests are *all* en route to LLC 5). When `load a` is executed, a memory request is sent to LLC 5, which takes 8 hops to traverse from core 15 to LLC 5. Upon receiving the request, the LLC partition services it (returns the data if the request hits in LLC; forwards it to memory channel if the request is a miss) and sends a response packet with the data back to the core taking another 8 hops. This is also the case with `load b`. In total, it takes 32 hops to request and get the data back for the two load instructions. Finally, a store request is sent which takes another 16 hops to store the data and receive an acknowledgment. Therefore, to compute `c[i]=a[i]+b[i]`, a total of 48 hops is required for the instruction sequence for each wavefront. Note that, the payload of the packets in the response traffic is considerably larger than request traffic. If we assign weights to the payloads with 1:5 ratio (req/ack:data), the total weighted hop count is 144.

Scenario 2: Let us assume a scenario, where all three memory requests go to different LLC partitions (Figure 3.3). In this case, `a` is mapped to LLC 5, `b` is mapped to LLC 6

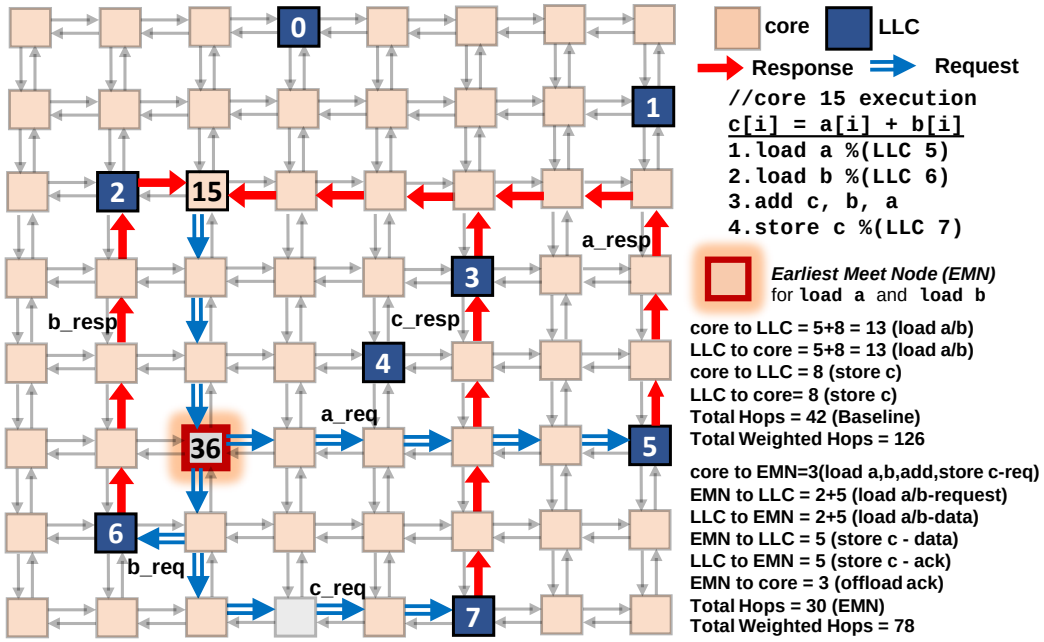


Figure 3.3: Earliest Meet Node for an instruction sequence (`c[i]= a[i]+b[i]`). For each memory operation, the request and response packets' traversal with YX routing is shown. All memory requests generate from core 15. The two loads and store head to LLC 5, LLC 6 and LLC 7, respectively. For this instruction sequence, the EMN is core 36.

and `c` is mapped to LLC 7. Getting the data for `load a` takes 16 hops and for `load b`, 10 hops. The `store c` is sent to LLC 7 and takes another 16 hops. Therefore, a total of 42 hops are needed for this computation. By considering the weights for each payload, the total weighted hop count is 126.

3.3.2 How to Reduce Data Movement?

To reduce data movement, we propose a new concept, called *Earliest Meet Node* (EMN). We observe that on the traversed path for both the load instructions, if we re-route the response messages with load data back to the request node in the same route as the request messages, there is a common node (LLC 5 in Scenario 1 and core 36 in Scenario 2 as shown in Figure 3.3) through which the data for both the loads pass, albeit not necessarily at the same time. This node is the EMN for the instruction sequence of a given wavefront.

Potentially, the computation (in this case – addition) can be performed at EMN and then, only an ack is sent back to the requesting core from the EMN on a successful computation.

For example, in Scenario 1, both the loads (along with the information of the store) can be sent as a single request (detailed in Section 3.4) from core 15 to the EMN (8 hops to LLC 5), and then the loads can be serviced at the LLC. Assuming the EMN can compute, the operation is performed once both the loads have been serviced. The store is then serviced by the same LLC, and an ack is sent back to core 15 (8 hops) indicating that the offload was successful. Therefore, the entire compute sequence requires 16 hops (reduced by 67%). Furthermore, the total weighted hop count reduces to 16 hops (reduced by 89%) as well. Similarly, for Scenario 2, shown in Figure 3.3, if both the loads and the store were sent as a single request (details in Section 3.4) to the EMN (3 hops to core 36), and then if the two loads are split and sent to their respective LLCs (5 hops (LLC 5) + 2 hops (LLC 6) = 7 hops), it would take a total of 10 hops. On their way back, rather than going back to core 15, both `a` and `b` can be sent to the EMN (5 + 2 = 7 hops). After computation, the result (`c`) can be sent directly to the LLC 7 (5 hops). The ack message for the store operation takes another 5 hops to reach the EMN. Finally, from the EMN, an ack notifying of a successful computation is sent to core 15 (3 hops). This approach would require a total of 30 hops (reduced by 29%) and a total weighted hop counts of 78 hops (reduced by 38%). *Note that, we do not change the routing policy (YX) or introduce any network deadlock when we re-route the response packets via the EMN.* We only decouple the response packets from its precomputed route to the core and instead send it to the EMN. This changes the route of the packet, while still following the routing policy. Based on this motivation, we propose a novel GPU design to dynamically build offload chains at runtime and make intelligent decisions for *opportunistically* offloading computations onto a location closest to where the required data is present, while minimizing data movement.

3.4 Opportunistic Computing

The main idea of the NDC mechanism is to first find candidate *offload chains* in a GPU application and to compute this chain *opportunistically* as close as possible to LLCs. To this end, we propose two schemes: **LLC-Compute** and **Omni-Compute**. The first scheme, **LLC-Compute** (Section 3.4.2), reduces data movement for offload chains for which the operands are found in the same LLC partition. An offload packet, which we call as *ComputePacket*, traverses to the destination LLC for computation and returns the result/ack back to the core. The second scheme, **Omni-Compute** (Section 3.4.3), is built on top of LLC-Compute and increases the coverage of operations, whose data movement can be further reduced, by enabling offloading for chains that request data from two different LLC partitions. In this case, the

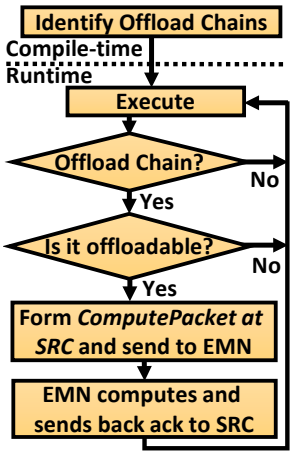


Figure 3.4: Key steps to realize computation offloading.

ComputePacket is sent to the EMN (another core), where the computation is performed. We discuss the process of finding the EMN in Section 3.4.3. If the data is placed on different LLCs with no common node in their YX/XY routes, then there is no EMN, and thus, computational offloading is ineffective. For example, in Figure 3.3, if core 15 sent two loads to LLC 0 and 6, there is no common node, and therefore, no EMN.

Figure 3.4 shows the steps in facilitating our proposed mechanisms: (1) identify offloadable chains by using compiler analysis to check for code patterns and register reuse and then tag the offloadable instructions; (2) during execution, efficiently identify and execute offload chain; (3) dynamically decide whether the computation can be offloaded or not; (4) form *ComputePackets* efficiently and keep track of offloaded chains; and (5) enable computation at EMN (LLC/cores).

3.4.1 What to Offload?

For applications with good L1 cache locality, offloading loads for computation without caching them locally would increase unnecessary data movement for these loads. Therefore, the ideal candidates for such computation offloading are the applications that are streaming in nature or have a long reuse distances that render the L1D cache ineffective. Note that, LLC-sensitive applications are also going to be beneficial, as our proposed mechanisms reduce the on-chip data movement between the cores and the LLCs.

It is well known that applications have different characteristics during their execution, and even if the entire application cannot be offloaded, certain phases/sequences of instruction from applications can still benefit from computation offloading. Therefore, rather than enabling computation offloading for the entire application, we propose to offload computation at the granularity of an instruction or a set of instructions, which could correspond to a high-level language statement such as the one shown in Figure 3.3. Note that, loads with further reuse at the core should not be offloaded, as it will negatively impact the L1D locality. Conservatively, we prioritize L1D locality over computation offloading. For this work, we target such instruction sequences that are amenable to computation offloading and are prevalent in many different types of applications such as machine learning kernels, linear algebra algorithms, cryptography, high performance computing applications and big-data analytics. Table 3.1 shows nine types of instruction sequences that we find amenable for offloading. *Note that more offload*

Table 3.1: Prevalent high-level code patterns along with their PTX instructions [1]. The response packet (type and size) details the amount of data that is sent back by the computation node to the source core.

Pattern	1	2	3	4	5	6	7	8	9
HLL code	c=f(a,b)	c=f(a,b,c)	c=a	c=f(a,c)	h(a,b)	h(a, i)	c=f(a,i)	d=f(a,d,g(b,i))	c=f(a,g(b,i))
PTX Code	ld a ld b alu c,a,b st c	ld a ld b alu c,a,b	ld a st c	ld a alu c,a st c	ld a ld b cmp a,b	ld a ld i cmp a,i	ld a ld i alu c,a,i st c	ld b ld i ld a alu c,b,i alu d,a,c	ld b ld i ld a alu c,b,i alu d,a,c st d
Response Packet	ack, 1 flit	data, 5 flits	ack, 1 flit	ack, 1 flit	bitmap, 1 flit	bitmap, 1 flit	ack, 1 flit	data, 5 flits	ack, 1 flit

i = immediate f(x,y),g(x,y) = arithmetic operator h(x,y) = logical operator

patterns can be formed using longer instruction sequences, but we concentrate only on patterns that can be packed into a single flit. Figure 3.5 shows the *ComputePacket* format for the largest sequence.

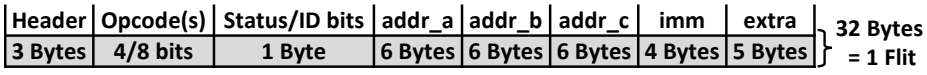


Figure 3.5: *ComputePacket* format for Pattern 9.

All nine patterns that we offload for computation start with a load instruction and end either with an ALU operation or a store operation. We call this sequence of instructions that can be offloaded as an *offload chain*. For this work, we tag the instructions in the applications that are potentially going to be offloaded at compile-time. These instructions can be easily found with a single compiler pass, and with multiple passes the compiler can even generate statically the data locality information [159] of the loads used by these instructions to make decisions for computation offloading. During execution, these tagged instructions get executed by the hardware, which selectively (based on the L1 locality) generates a single packet for computation offloading.

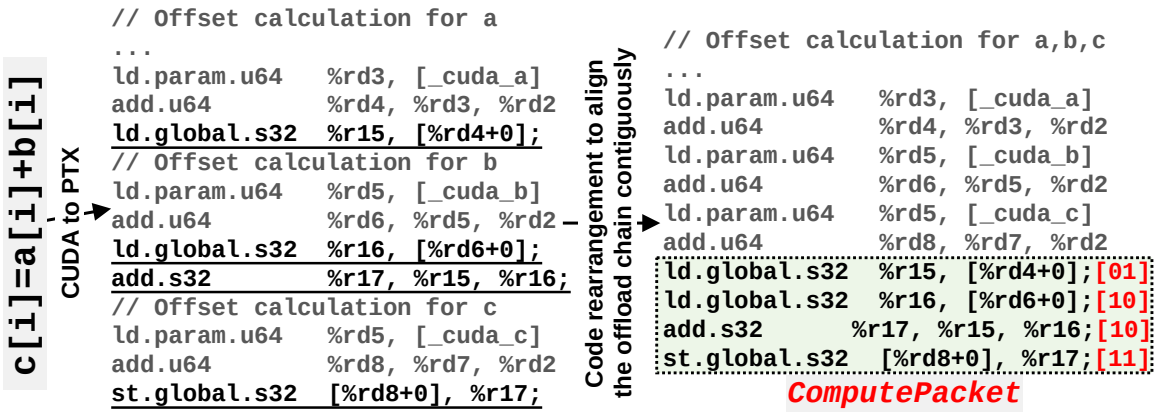


Figure 3.6: Representative code snippet. The offload chain is tagged and rearranged in the PTX code to align contiguously in memory.

3.4.2 LLC-Compute

LLC-Compute offloads chains where *all* the operands are headed towards the same LLC partition. We now describe the steps of LLC-Compute design.

Identification of Offload Chains: In order to ensure that offloading is not performed in the presence of high L1 locality, we use compiler analysis to identify the offload chains that are expected to improve performance if offloaded. The compiler identifies offloadable instructions based on their "per-thread" reuse patterns. Note that our approach does not take into account the reuse distance of the memory request. If it finds no register reuse of the memory request by the same wavefront, it will tag it for offloading.

We analyze the Parallel Thread Execution (PTX) ISA [1] generated by the CUDA compiler [92]. By default, each load/store instruction in the PTX code is preceded by its effective offset calculation that is needed for address generation. We demonstrate this with an example in Figure 3.6, which shows a code snippet in high-level language and its corresponding PTX instructions. First, offset for `a` is calculated, and then `a` is loaded. The case is similar for `b`. In our approach, as shown in Figure 3.6, we modify the compiler to transform the PTX code so that the offset calculations for the loads/store for the offload

chain are executed earlier, and the offload chains' instructions are contiguously stored. This reduces the processing time to form a *ComputePacket*. Also, similar to prior work such as TOM [76], our compiler tags the opcodes of the offloadable instruction sequences with two bits to indicate the first, intermediate and the last instructions in the offload sequence. Specifically, we tag the first load with the bits `01` and then the intermediate PTX instructions with `10` until the final instruction, which is tagged as `11` indicating the end of the chain. Furthermore, these tags also allow for efficient wavefront scheduling that is computation offloading aware, as discussed later in this section.

Hardware Support for Offloading: Figure 3.7 shows the required hardware changes (in black) to the baseline architecture for our two proposed mechanisms. It also shows the connections needed for LLC-Compute and Omni-Compute to be integrated with the GPU design. Figure 3.8 provides the detailed implementation of the components (Offload Queue and Service Queue). We first describe the hardware additions needed for LLC-Compute. To enable offloading, we add an additional component called the Offload Queue (OQ) (❸), which is responsible for generating, offloading, and maintaining the offloaded chains status. As shown in Figure 3.8(a), OQ is made up of three components: Offload Queue Management Unit (OQMU) (❹), Offload Queue Status Register (OQSR) (❺), and *ComputePacket* Generation Unit (CPGU) (❻). The OQMU is the controller unit. It (1) decides whether to offload computation or not based on EMN computation and L1 locality, (2) initiates computation offloading, (3) manages the OQSR, and (4) receives the result/ack for offloaded computation. The OQSR is a 48-entry (Section 3.6.2) status register to maintain the status of the offloaded chains. CPGU is responsible for generating a *ComputePacket* with the computed EMN based on both the load requests' LLC partitions and injecting it into the network for transmission. We give a detailed explanation of how these components are utilized in Section 3.4.4.

Efficient ComputePacket Formation: While the OQ is responsible for generating *ComputePackets*, it has no control on how instructions are fetched and decoded, and how

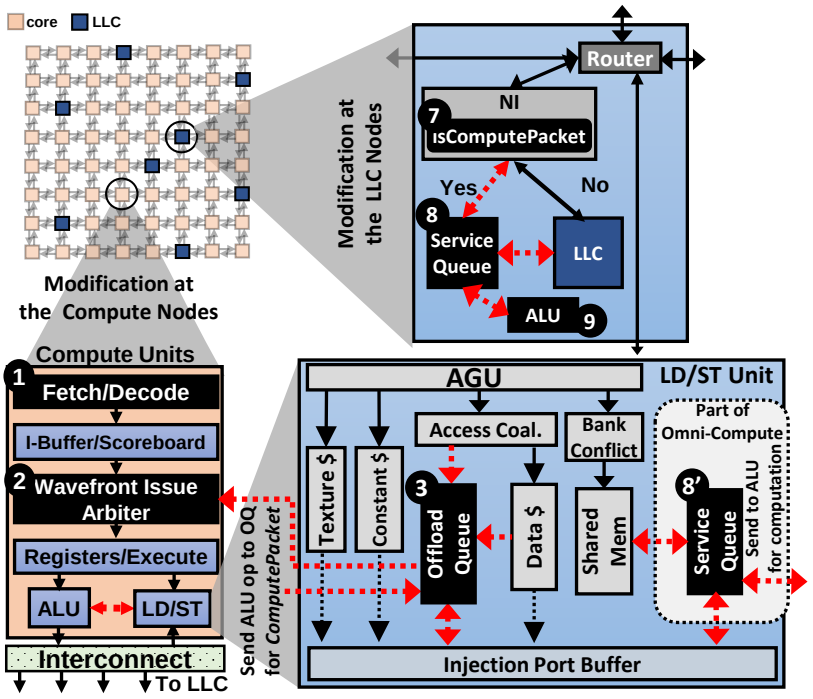
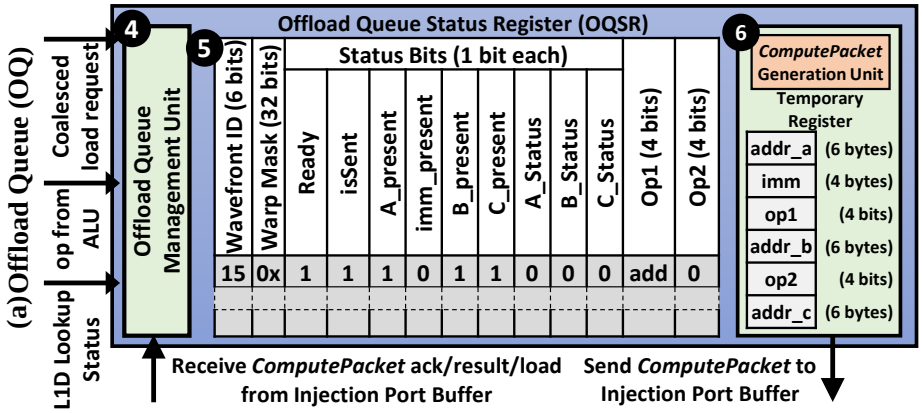
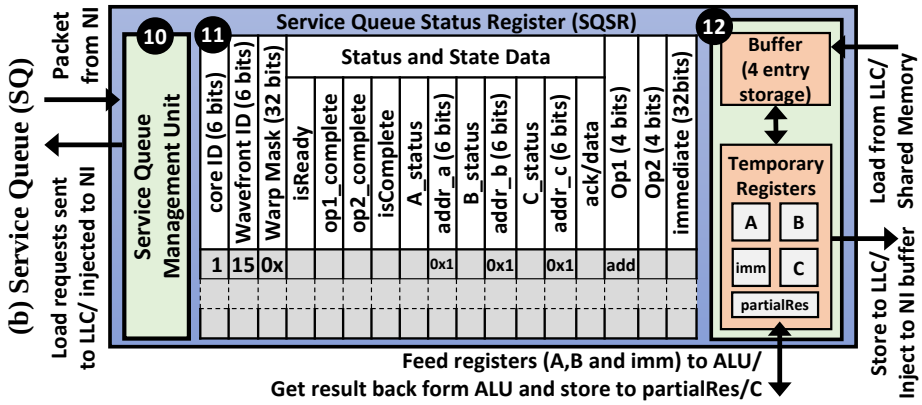


Figure 3.7: Proposed hardware modification to enable offloading. Additional/modified units are shown in black; The additional unit in Omni-Compute (over LLC-Compute) is the SQ in LD/ST unit 8'.

wavefronts are issued. Therefore, relying only on the OQ for generating *ComputePackets* is not optimal in terms of performance due to two reasons. First, due to limited instruction buffers (Section 3.2), not all the instructions in an offload chain can be issued close in time. Furthermore, instruction fetch and decode takes place in a round-robin fashion for all the wavefronts, thus, leading to large queuing delays for issuing any remaining instructions in an offload chain of a given wavefront. Second, due to the baseline wavefront scheduling policy (Section 3.2), each load in an offload chain (that results in a cache miss) will cause the wavefront to be switched out for another wavefront. Therefore, in order to issue two loads from the same wavefront, many other wavefronts get executed. This leads to partially filled OQSR entries for many wavefronts. Only when all the offload chain instructions of a given wavefront are executed, a *ComputePacket* is formed. This leads to longer latencies in creating *ComputePackets*. Moreover, the CPGU



(a) Offload Queue.



(b) Service Queue.

Figure 3.8: Hardware design of the additional components to support computation offloading.

would need to maintain buffers for each partial *ComputePacket* leading to higher overheads.

To mitigate the effects of wavefront scheduling and avoid the overheads of implementing a CPGU with large buffers, we modify the wavefront scheduling policy (Figure 3.7 ❷) along with the instruction fetch and decode logic (Figure 3.7 ❶) to prioritize *ComputePacket* formation. We achieve this by making the instruction tags in offload chains known to them. On the first instruction fetch of a wavefront with the tag `[01]`, we prioritize the fetch and decode of this wavefront over other wavefronts. Therefore, whenever a single entry in the

instruction buffer for this wavefront becomes empty, the fetch unit prioritizes the fetch for this wavefront over other wavefronts and decodes it and stores it in the available buffer space. Similarly, on the wavefront scheduler logic, we prioritize the wavefront that is issuing the offload chain. When the final instruction (tagged `[11]`) in the offload chain has been fetched, decoded and issued, the fetch, decode and wavefront scheduling logics fall back to their default scheduler logic. This reduces the latency for the creation of a *ComputePacket*, enabling better injection into the interconnect by interleaving the generation and injection of the *ComputePackets*. This also minimizes the storage overheads for the CPGU, as it only needs to maintain the full offload chain information for only one wavefront at any given moment. Once an instruction in the offload chain is added to the OQSR, it is treated as committed, after which other instructions in the offload chain are issued and executed, allowing the formation of *ComputePackets*. Only the final instruction in the offload chain will cause the wavefront to stall and wait for an ack.

Hardware Support for Enabling Computation at the LLC Partitions: To enable offloading to LLC partitions, we need to add computation unit, logic to decode *ComputePackets*, and status registers for bookkeeping of the offloaded computations to the LLC partitions. The first addition is a 32-wide single-precision floating point and integer ALU (Figure 3.7 ❾) in order to compute the offload chains. We keep it to be 32-wide to maintain the same ALU latency as to that of a core. The second addition is a multiplexer on the network interface (Figure 3.7 ❼). The multiplexer directs the packet to the Service Queue or the LLC based on whether the header bit identifies it as a *ComputePacket* or not. The third addition is a component called the Service Queue (SQ) (Figure 3.7 ❽), which further comprises of three units as shown in Figure 3.8(b): Service Queue Management Unit (SQMU) (Figure 3.8(b) ❿), Service Queue Status Register (SQSR) (Figure 3.8(b) ⓫), and a temporary buffer (4 entries) (Figure 3.8(b) ⓬) to queue up multiple offload chains for computation. The SQMU decodes the received packet, updates the SQSR entries, and generates the required load requests to send to the LLC. SQSR is a 96-entry (Section 3.6.2) status table, and it maintains the addresses of

the load and store requests of the offload chains. The SQSR then sends the load requests to the LLC and once data is available in the LLC, the LLC sends signals to the SQ to update the availability bit for the offload chain in the SQSR[1]. Note that, if the load request resulted in an LLC miss, we do not differentiate between the regular memory requests and the memory requests from an offload chain at the DRAM request scheduler. Prioritization techniques can be employed to improve the performance further, but we leave this as potential future work. A temporary buffer holds the offload chains' data. The buffer entries are filled on a first-come first-serve basis for the offload chains. The buffer fills the gap between the time it takes to read required data from LLC and bring them to SQ to feed to the ALU for computation. By having ready entries in the buffer, we can hide the latencies for other requests to be serviced and fetched from LLC into the buffer. If the buffer is full, no new loads are fetched from the LLC. Once an entry is ready to be computed, the data is moved from the buffer to the temporary registers, which are then fed to the ALU. Every cycle, if required, the buffer fetches the required data from LLC based on the status bits of the SQSR entry (to make sure it is present in the LLC), to maintain a set of compute-ready data. Once the computation is finished, SQMU removes the entry from SQSR. It then generates and injects an appropriate response packet to send to the core based on the type of offload chain.

3.4.3 Omni-Compute

As discussed in Section 3.3, EMN for an offload chain need not be an LLC partition. Thus, we propose Omni-Compute, which adds support for offloading computations to other cores, which are the EMN for a given offload chain.

Unlike LLC-Compute, finding the EMN in Omni-Compute is not straight forward because the required data is present in different LLC partitions. Algorithm 2 details the

[1]We do not send data from the LLC until it is needed for computation, which is initiated when all the required operands are present in the LLC. By exploiting the LLC to store data, we avoid any data storage overheads.

mechanism that is used to find the EMN (example in Figure 3.3). For a given GPU configuration, a simple lookup table can be populated using Algorithm 2 and be used to find the EMN during GPU execution. The basic idea for finding an EMN is to find a node that is common in the traversal paths (XY or YX routes) of the two loads. If there are multiple common nodes in their paths, EMN is the node that is *closest to both* the LLC nodes.

In order to provide the ability to offload *ComputePackets* to any node, we add a Service Queue (SQ) to all the cores (Figure 3.7 ❽'). The SQ used in the cores is different from the one used in LLC partitions only in its input/output connections. Rather than sending load requests directly to the LLC queue, the SQ injects the load requests into the interconnect for transmission to their respective LLC partitions, receives the data from the LLCs, and updates the SQSR accordingly. In LLC-Compute, the SQ uses the LLC as a storage for the loads, whereas, in Omni-Compute, for the SQs in the cores, we reuse the shared memory as the storage structure to buffer the loads as they are received. Once the entry is ready to be computed, the data is transferred from the shared memory to the temporary registers, which are fed to the ALU of the core. Note that SQ only reuses the ALU when it is idle and stalls if it is in use by the core/wavefront scheduler.

As Omni-Compute enables computation at any node, a higher number of offload chains can find EMNs and therefore be offloaded. This may leave core resources under-utilized. On the other hand, as Omni-Compute offloads computation to other cores, the SQs at these cores will make use of the ALUs and improve core utilization. Note that the computations and bookkeeping are done by the GPU core. Therefore, we do not make any modification to the router/interconnect.

Algorithm 2 Finding EMN for 2 loads

```
INPUT:  core_node                  // Denotes the offloading GPU core node
        llc_nodes[0/1]             // Denotes the destination LLC nodes for the loads
 1: if llc_node[0] == llc_node[1] then                          //LLC-Compute scenario
 2:     return llc_node[0].
 3: end if
 4: for i ← 0 to 1 do
 5:     traversal_path[2i] = xy_path(core_node, llc_node[i])
 6:     traversal_path[2i + 1] = yx_path(core_node, llc_node[i])
 7: end for
 8: for i ← 0 to 1 do
 9:     for j ← 2 to 3 do
10:         common_nodes.append(traversal_path[i] ∩ traversal_path[j])
11:     end for
12: end for
13: for nodes n in common_nodes do
14:     dist = manhattan_dist(n, llc_node[1]) + manhattan_dist(n, llc_node[2])
15:     if dist < min_distance then
16:         min_distance = dist;
17:         EMN = n
18:     end if
19: end for
20: return EMN.
```

3.4.4 How Does Our Mechanism Work?

In this section, we describe how LLC-Compute and Omni-Compute work under multiple scenarios.

3.4.4.1 Mechanism to Offload Computation

Let us refer to Figure 3.6, when the first load instruction (tagged `[01]`) is fetched and decoded. The wavefront that the instruction belongs to is prioritized for fetch and decode until the final instruction in the offload chain that belongs to the same wavefront is fetched and decoded. This wavefront is then de-prioritized. Similarly, when the first instruction gets issued by the Wavefront Issue Arbiter, that wavefront gets prioritized for issuing. Starting from this point, we show how computation is offloaded using Figure 3.9. The Wavefront Issue Arbiter issues the first tagged instruction in an offload chain to the LD/ST unit, which forwards it to OQ (①). There could be three possible scenarios after the first load is issued.

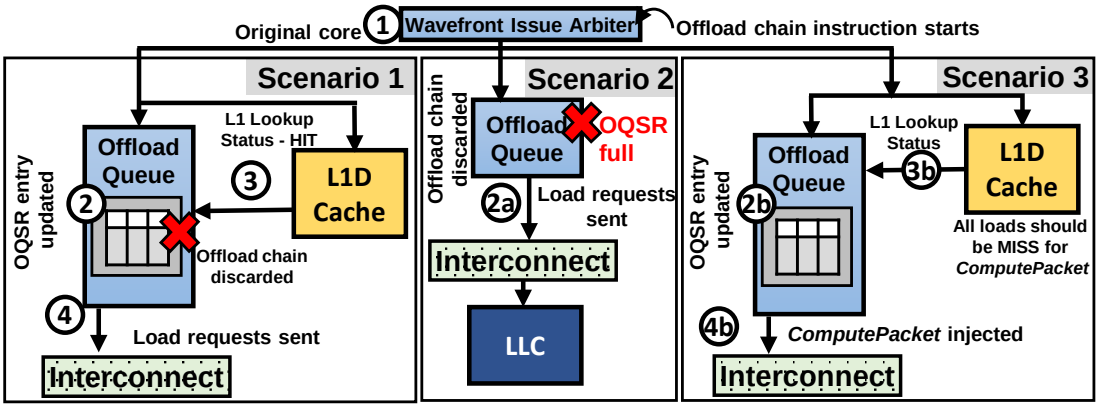


Figure 3.9: Scenarios for computation offloading.

Scenario 1: L1 Lookup is a *hit*: The instruction is sent to the OQ and the OQSR is updated (②). Simultaneously, an L1D lookup is performed and the status is returned to OQ (③). In this case, the L1D lookup was a *hit*. This would cause the offload chain to be discarded and regular execution of the instructions to resume. If the load being executed was a second load in the sequence (assuming the first load was a miss), and this was a *hit* in the L1D cache, the OQSR entry is flushed and the first load is sent to the LLC partition (④). Note that, the first load does not execute again but does incur a 1-cycle latency to generate the packet and push into the interconnect. This additional cycle amortizes as the memory requests being fetched from LLCs take 100s of cycles.

Scenario 2: OQSR is full: When the first load in the offload chain is sent to the OQ, and OQSR is full, the offload chain is discarded and the wavefront resumes normal execution (②a). The second load cannot cause this as the first load would have either reserved an entry in OQSR or had been discarded.

Scenario 3: ComputePacket is formed: When all the loads are misses in the L1D cache (③b), if the computed EMN is compute capable, and when the final instruction in the offload chain is issued and the OQSR is filled (②b), a *ComputePacket* is formed and

injected into the network (④b) and the wavefront is stalled until the EMN sends back the result/ack to continue execution. If the EMN is not compute capable, the OQSR entry is flushed and the loads are sent individually. For example, in LLC-Compute, only the LLC nodes are capable of computing the offloaded chains.

3.4.4.2 Mechanism to Execute Offloaded Computation

When a packet is ejected from a router, a header bit is checked to determine if it is a *ComputePacket*. If it is, then it is sent to the Service Queue (SQ) (Figure 3.10 ⑤). Figure 3.10 shows three possible scenarios when a *ComputePacket* is received.

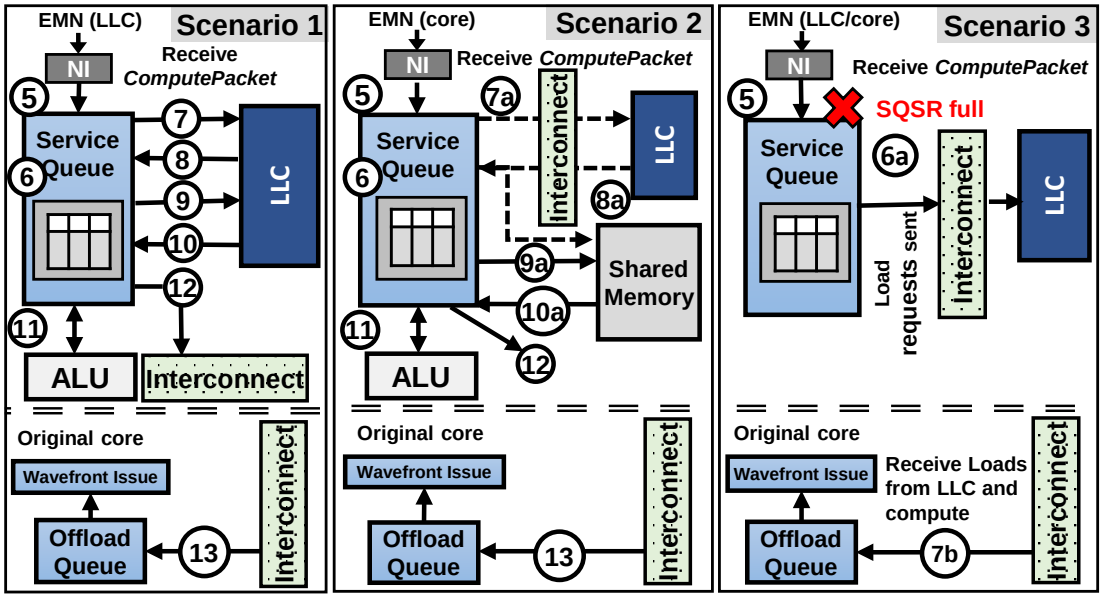


Figure 3.10: Scenarios when *ComputePacket* is received.

Scenario 1: EMN is an LLC partition: When a *ComputePacket* is received, the SQMU decodes the packet, fills a new entry in the SQSR and updates the respective bits (⑥). Two loads for `a` and `b` are sent to the LLC partition (⑦). When any of the loads is serviced by the LLC (once it is available in the cache), an update message from the LLC is received (⑧), and the status bit for the particular load is set. When both the loads are available,

the Ready bit (see Figure 3.8 (❺)) is set. Then, the SQ sends load requests to the LLC (⑨), and the data is brought into the buffer ⑩. If the buffer is full, the SQ stalls until the buffer has an empty entry. Once the entry is ready in the buffer, the entry is popped and sent to the temporary registers which transmit the data to the ALU for computation and receive the results back ⑪. Once the result is received, based on the status bits, the store is forwarded to its corresponding LLC partition and an ack is sent to the core that offloaded the computation (⑫). When the core receives the ack, the wavefront resumes regular execution and proceeds to remove the entry from its OQ (⑬).

Scenario 2: EMN is a compute capable core: The computation offloading process at a core is similar to offloading to LLC. The only difference being, the SQ receives the *ComputePacket*, requests for the loads from the respective LLCs (⑦a), fetches (⑧a) and stores them in shared memory (⑨a). The shared memory location is stored in the SQSR rather than the memory address. Similarly, when both the loads are received, the data is fetched from the shared memory to the buffer (⑩). The rest of the process is similar to Scenario 1.

Scenario 3: SQSR at EMN is full: When the SQSR is full (either the ones at the LLC or core), upon a *ComputePacket* being received by the SQ (⑤), the SQMU generates two loads and send them to LLC for servicing (⑥a). It tags the memory requests as non-offloadable, causing the memory requests to go back to the core that offloaded them. These loads will reach the core (⑦b), and the OQMU will check if this was an ack or not. Upon finding a load rather than an ack, the OQSR entry appropriately updates itself to highlight the status of the loads (a_present, b_present). Once the computation is done, the entry is removed from OQSR, a store request is sent, and the wavefront resumes regular execution.

3.4.5 Limitations of Computation Offloading

In this work, we only consider offload chains whose loads are cache miss to preserve as much locality as possible. Furthermore, the issues of address mapping and data placement play a big role in whether an offload chain can be computed at the EMN or not. For example, due to the LLC placement, there can be offload chains with two load requests without any overlapping nodes during their NoC traversal, and therefore, no computation offloading is performed. Additionally, due to the lock step execution of wavefronts (Section 3.2), applications with high degree of control-flow and irregular access patterns may lead to control-flow divergence and memory-level divergence, respectively. This can cause significant amount of different computations to take place for different threads in a wavefront and multiple memory requests to be generated per wavefront, respectively. This would result in multiple *ComputePackets* from a single wavefront to be generated, leading to higher bookkeeping overheads. In case of only memory divergence, the *ComputePacket* is generated such that each instruction can only generate a single memory request (threads that require other data will be inactive in the warp mask). Currently, for wavefront divergence, we have warp mask bits in OQ and SQ for each entry, while we handle memory divergence by passing the mask bits in another flit attached to the *ComputePacket*. Divergence can be mitigated by smarter data and compute placement [160] or by efficiently forming wavefront dynamically [161]. However, we do not explore such optimizations, but rather provide the hardware design for enabling computation offloading. Also, as shared memory is used by the core and the SQ, we have to make sure it is not overprovisioned. Conservatively, we offload only in situations where the shared memory is large enough to accommodate the needs of both core and SQ. During compilation, we check the shared memory usage in the kernel and disable computation offloading if needed.

Need for Hardware Optimizations: The end result of reducing data movement can also be achieved using compiler analysis to dynamically compute the desired indices (thereby

changing the required data) for the threads to work as shown in [162]. However, compiler-based optimizations rely heavily on static analysis and cannot adapt themselves to dynamic behavior of applications/runtime parameters. For example, finding EMN at compile time for all the computations requires prior knowledge of address mapping, data placement, compute placement (thread-block and wavefront scheduling), and other architecture specific parameters. Most of these parameters are not exposed to the compiler, and also change dynamically during runtime (e.g., data migration, compute placement, etc.). Not having all the required *a priori* knowledge will make the analysis of the compiler incomplete. Therefore, it will not completely optimize the computation offloading for reducing data movement. Also, other hardware optimizations such as forming macroinstruction for the offload chains can be performed. But, using macroinstruction will lead to larger hardware overheads as multiple memory requests will need to be decoded concurrently. Therefore, handling a macroinstruction will require multiple fetch/decode/load-store units. If request generation is serialized, it effectively becomes similar to our scheme.

3.5 Experimental Methodology

Simulated System: We simulate the baseline architecture as mentioned in Table 3.2 using GPGPU-Sim v3.2.2 [103]. We extensively modified GPGPU-Sim to implement our proposed schemes. OQ and SQ were added to the GPU datapath and integrated into the core model. We add an ALU and a Service Queue to the LLC datapath. We model the GPU cores, cache and DRAM power using GPUWattch [88]. Based on the injection rate obtained from the simulations, we configure DSENT [163] to find the average power of the interconnect. We run the applications until completion or 1 billion instructions, whichever comes first. The applications and their inputs are large enough to fill the workgroup slots.

Benchmarks: We simulate multiple microbenchmarks that are commonly present in many applications such as scientific computing, machine learning, linear algebra, big data, etc.

Table 3.2: Configuration parameters of the GPU.

GPU Features	1.4GHz, 56 cores, 32 SIMT width, GTO wavefront scheduler
Resources/Core	48KB shared memory, 64KB register file, Max. 1536 threads (48 wavefronts, 32 threads/wavefront)
Private Caches/Core	16KB L1 D-cache, 12KB T-cache, 8KB C-cache, 4KB I-cache, 128B block size
L2 Cache	0.5MB/Memory Partition, 16-way 128 KB Line size
Memory Model	8 MCs, FR-FCFS, 8 banks/MC, 1000 MHz Partition chunk size 128 bytes, 500 GB/s peak BW
GDDR5 Timing	$t_{CL} = 11$, $t_{RP} = 11$, $t_{RC} = 39$, $t_{RAS} = 28$, $t_{CCD} = 2$ $t_{RCD} = 11$, $t_{RRD} = 5$, $t_{CDLR} = 5$, $t_{WR} = 12$
Interconnect	8×8 2D Mesh, 1400MHz, YX Routing, 1 core/node, 8VCs, flit size=32B, Buffers/VC=8, islip VC & switch allocators

Table 3.3: List of evaluated benchmarks. The patterns listed here correspond to the patterns in Table 3.1.

Micro-benchmark	Pattern
COMPARE	5
COPY-ALIGNED	3
COPY-STRIDED	3
DENSITY	6
VECADD-ALIGNED	1
VECADD-STRIDED	1
NORM	7

Workload	Pattern	Suite	Dyn.Inst.	Mem.Req.
BFS	2,6,7	CUDA	18%	47%
FDTD	1,2	PolyBench	7%	67%
KMN	2,4	Rodinia	23%	67%
MVT	1,2	PolyBench	25%	69%
RED	1,2	SHOC	16%	75%
SCP	2	CUDA	12%	67%
SRAD	1,2,3,7	Rodinia	3%	75%
STREAM	2,8	Rodinia	33%	67%
TRIAD	9	SHOC	18%	60%

We also analyze 9 GPGPU workloads from SHOC [87], PolyBench [93], CUDA SDK [92] and Rodinia [94] benchmark suites. Table 3.3 lists the evaluated microbenchmarks and workloads. The microbenchmarks include: `COMPARE`, which performs point-wise comparison of two strings to count the number of different elements; `COPY-X`[2], which copies one array into another array; `DENSITY`, which counts the number of 0's in an array; `VECADD-X`, which sums two 1-D arrays and stores the result in a third array; and `NORM`, which normalizes a 1-D array with a given value. To highlight the significance of the instructions that we optimize for, we show the fraction of dynamic instructions and total memory requests (Table 3.3) that can be offloaded. On an average, 17% of dynamic instructions and 66% of memory

[2]X is either ALIGNED or STRIDED, indicating whether the memory addresses belong to same or different LLC nodes, respectively.

requests are tagged as offloadable.

The proposed techniques in this work require global memory accesses and are ineffective towards applications that are optimized using GPU features such as shared memory, constant caches, etc. This does not necessarily mean that the scope of the proposed techniques is reduced. Rather, by rewriting applications to make use of computation offloading, it is possible to achieve better energy efficiency/performance. Hence, as a proof of concept, we modified the `RED` application that relies on shared memory to compute partial sums to make use of global memory. We further develop a hybrid version of the workload that combines the global memory and shared memory approaches to find a sweet spot in energy efficiency/performance (Section 3.6.2). Similarly, applications using fused-multiply-add (fma), which require 3 different loads, were broken down into multiply and add instructions, and only the multiply instruction with 2 loads is offloaded and the result is sent back to the core for accumulation with the reused third load. However, our baseline execution results use all the applications' unmodified version.

Hardware Overheads: We implement a 48-entry OQSR in OQ and a 96-entry SQSR in SQ, which was empirically decided as discussed in Section 3.6.2. The OQ also consists of OQMU logic and 30 bytes of storage that is needed for *ComputePacket* generation, but the largest component is the OQSR. OQSR needs roughly 330 bytes of storage. Considering all the register/buffer overheads in SQ, it requires approximately 2.4kB of storage. Current generation GPUs have L1 caches as large as 128kB in their cores [25], indicating that the area overheads are negligible. We use CACTI [164] to model the additional structures and integrate their leakage and dynamic power requirements in our experimental results. For the additional ALUs in the LLCs, we assume it to be similar to the power and area of an SP unit in a core, which is modeled using GPUWattch [88]. To find the EMN at runtime, a lookup table of 64 entries (8×8 combinations) is stored at each core that uses MC ids to find the EMN. All the EMNs for each combination of core and MCs are computed statically (using Algorithm 2) as the routing paths are static. The additional ALUs, EMN lookup

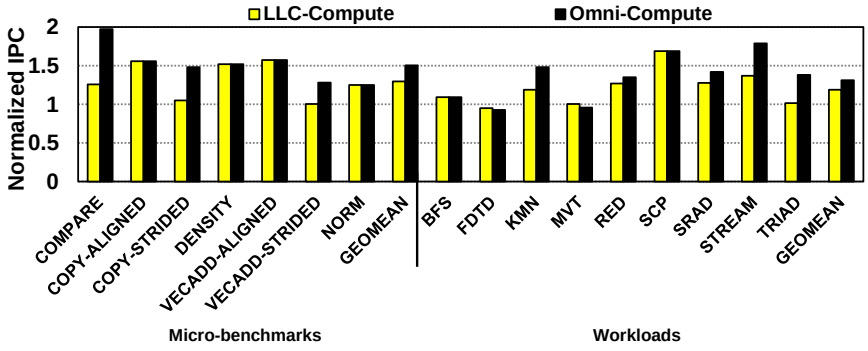
tables, and all the OQs and SQs require less than 1% area of a high-performance GPU.

3.6 Experimental Results

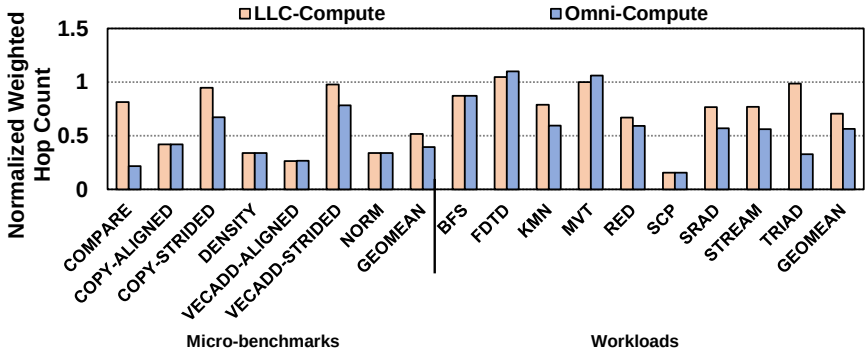
To evaluate the benefits of the proposed schemes, we measure the GPU performance (instructions-per-cycle (IPC)), energy efficiency (performance/watt), normalized average memory latency and reduction in on-chip data movement (weighted hop count). The weighted hop count is the sum of all link traversals by flits, which indicates the total amount of on-chip data movement. The average memory latency is the round-trip latency of all the memory requests (L1 misses) in an application. It is made up of the service delay and NoC delay. Service delay is the time the LLC/DRAM takes to access the memory and service the request. The NoC delay consists of the traversal delay, the queuing time at the injection/ejection buffers of the cores/LLCs. All results are normalized to the execution of unmodified workloads on the baseline GPU without computation offloading.

3.6.1 Effects of Proposed Mechanisms

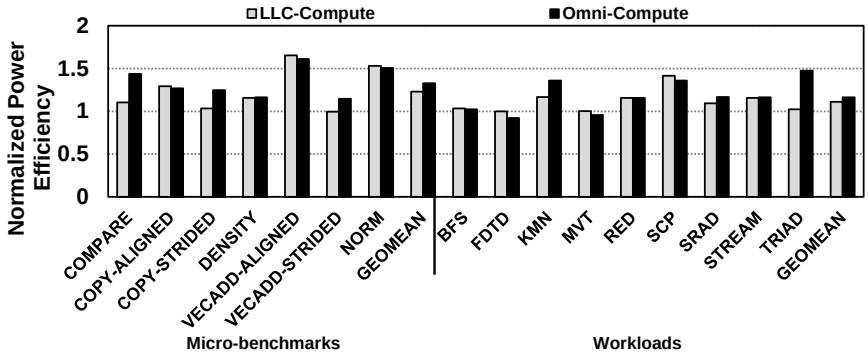
Effects of LLC-Compute: Figure 3.11 shows the performance, performance/watt and the weighted hop count benefits of our LLC-Compute mechanism for seven microbenchmarks and nine workloads. LLC-Compute increases the performance and performance/watt for the workloads by 19% and 11%, respectively. It also reduces the weighted hop counts for the workloads by 29%. For the microbenchmarks, it achieves performance and performance/watt improvement of 30% and 23%, respectively. Figure 3.12 shows the percentage of offloadable chains that were offloaded by the proposed mechanisms. As mentioned in Section 3.4.5, not all offload chains can be offloaded due to their data placement, cache behavior and/or due to the lack of free slots in OQ/SQ. For applications such as `COPY-ALIGNED`, `DENSITY`, `VECADD-ALIGNED`, `NORM`, `RED` and `SCP`, LLC-Compute is able to improve performance immensely. This is due to the high amounts



(a) Performance.



(b) Weighted Hop Count.



(c) Power efficiency.

Figure 3.11: Impact of proposed mechanisms.

of offloading as seen in Figure 3.12. Applications such as `COMPARE`, `STREAM`, `KMN` and `SRAD` achieve modest gains due to the relatively less amount of offloading. The performance gains achieved by offloading can be correlated with the reduction in average memory latency of the packets as shown in Figure 3.13. Note that, Figure 3.13 is the detailed version of Figure 3.2(b). On an average, for the workloads and microbenchmarks, the

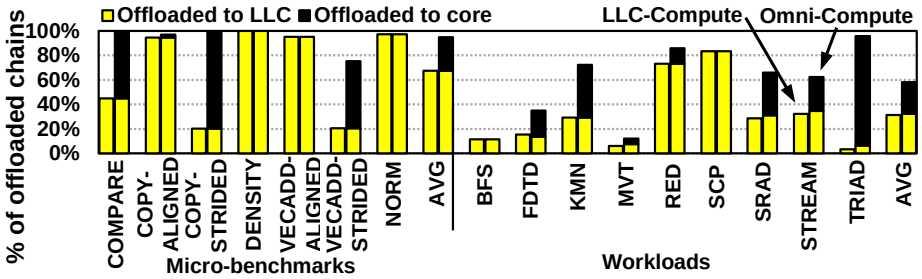


Figure 3.12: Percentage of offloaded chains.

average memory latency reduces by 16% and 29%, respectively.

For applications like `COPY-STRIDED`, `BFS`, `FDTD`, `MVT`, `TRIAD` and `VECADD-STRIDED`, we see that LLC-Compute is not able to improve performance. Rather, in the case of `FDTD`, the performance is slightly reduced and the weighted hop counts slightly increased. `FDTD` and `MVT` do not show improvements because of good L1 locality, which leads to less offloaded chains as seen in Figure 3.12. The small amount of offloaded chains also causes slight increase in L1 misses, thereby counteracting any benefits in case of `MVT`, but slightly reducing performance and increasing the data movement for `FDTD`. For `COPY-STRIDED`, `VECADD-STRIDED` and `TRIAD`, the reason behind the lack of benefits is due to the lack of offload chains that can be computed at the LLCs. In `BFS`, there are many offload chains, but due to wavefront and memory divergence, the number of in-flight offload chains (each wavefront generates multiple offload chains) is much larger than what OQ and SQ can service.

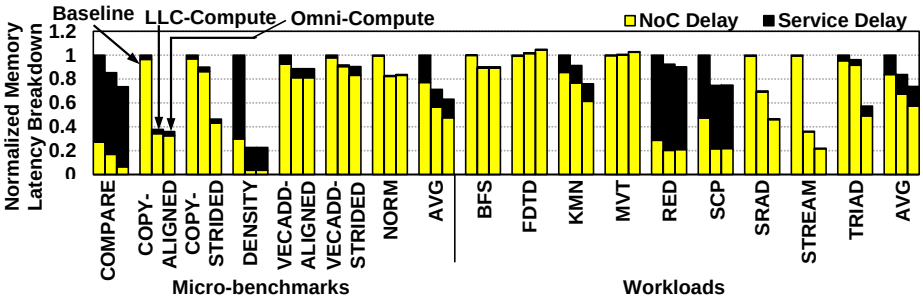


Figure 3.13: Percentage reduction and breakdown of average memory latency.

Effects of Omni-Compute: Figure 3.11 also shows the simulation results for our

Omni-Compute mechanism. With Omni-Compute, performance and performance/watt for the workloads increase on an average by 31% and 16%, respectively. For the microbenchmarks, it improves the performance and performance/watt by 51% and 33%, respectively. It also reduces the weighted hop counts by 44% for the workloads and by 61% for the microbenchmarks. Figure 3.12 shows the percentage of offloadable chains that were offloaded by the proposed mechanism. As shown in Figure 3.13, on average, the average memory latency for the workloads and microbenchmarks reduces by 27% and 37%, respectively.

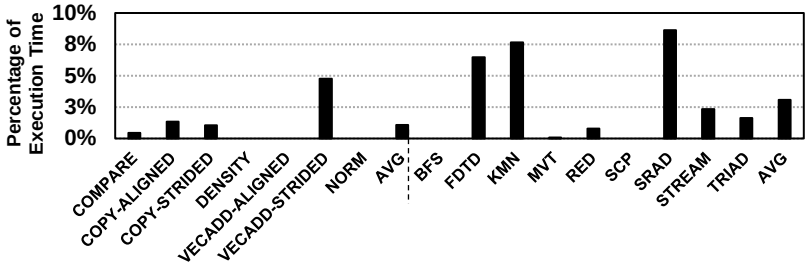


Figure 3.14: Percentage of execution time when either the core or the SQ contend for ALU.

As Omni-Compute builds on top of LLC-Compute, it allows for computation offloading at other GPU cores as well. We see that applications such as `COMPARE`, `COPY-STRIDED`, `VECADD-STRIDED`, `KMN`, `SRAD`, `STREAM` and `TRIAD` improve greatly when compared to LLC-Compute. This is because of the additional opportunities for computation offloading to other cores that is available to Omni-Compute. Applications such as `FDTD` and `MVT` suffer even more than LLC-Compute due to a reduction in L1 locality due to the increase in computation offloading. Note that the cache behavior is dynamic and by offloading computations, we do not get the load requests back to the core, thereby, changing the access pattern. Applications such as `COPY-ALIGNED`, `DENSITY`, `VECADD-ALIGNED`, `NORM`, `BFS`, `SCP` and `RED` do not improve much when compared to LLC-Compute because of their respective data placement. Most of the offload chains are already offloaded to LLCs, making LLC-Compute good enough. Figure 3.14 shows the percentage of execution time when either the core or the SQ is in contention for the ALU.

In applications such as VECADD-STRIDED, FDTD, KMN and SRAD, the contention for the ALU is relatively high compared to other applications. This is due to the fact that not all offload chains are offloaded and are left for the core to execute. Also, in SRAD and KMN, there are many compute instructions apart from offload chains. This causes the SQ to contend for ALU while the core is using it.

3.6.2 Sensitivity Studies

Interconnect Topology: We evaluated Omni-Compute with multiple interconnect topologies: butterfly (4-ary, 3-fly), crossbar (56×8), and mesh (8×8). We also evaluated them with double their bandwidth. Figure 3.15 shows the impact of using

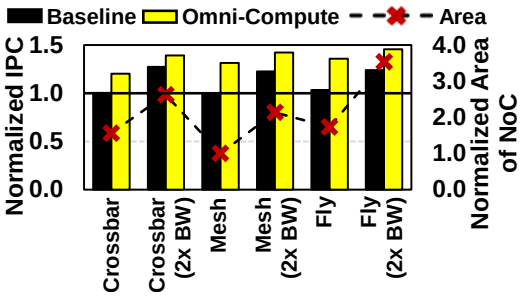


Figure 3.15: Impact of interconnect topology on performance and area.

Omni-Compute towards average performance across all the workloads for each of the topologies. On an average, the performance of crossbar and mesh remain similar while the performance improves by only 3% on butterfly topology. This is due to the large queuing delay at the injection ports at the MCs as observed in Figure 3.2(b). Even with decreased hop count of butterfly and crossbar, they do not affect the overall latency of the requests by much. With Omni-Compute, the performance of crossbar and butterfly improves by 20% and 36%, respectively. The benefits in crossbar are not as much as butterfly and mesh. This is because, in crossbar, computations can only be offloaded to LLC nodes, while in butterfly, all the routers and LLC nodes are capable of computing. Furthermore, butterfly is able to achieve a higher performance compared to mesh as it is able to offload

chains that were not offloadable in baseline (mesh) due to its dimensional routing policy. Note that these improvements are due to the reduction in NoC delay similar to mesh in Figure 3.13. Furthermore, we also doubled the bandwidth of mesh, crossbar and butterfly and found that the performance improves by 27%, 23% and 24%, respectively. With Omni-Compute, it further improves to 39%, 42% and 46%, respectively. This highlights the fact that the on-chip bandwidth is a bottleneck in GPUs and doubling the bandwidth is still not sufficient to eliminate all the congestion delays as we still achieve (albeit relatively lower) performance improvements with Omni-Compute. Figure 3.15 also shows the normalized area of using these topologies.

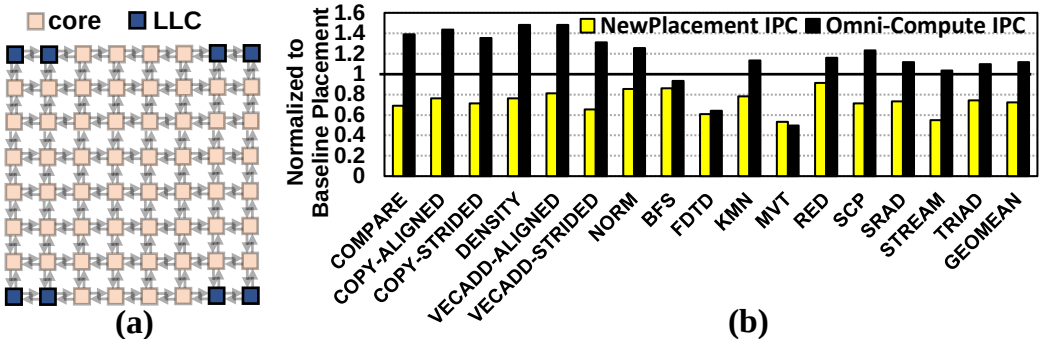


Figure 3.16: Impact of LLC placement. (a) LLC placement, (b) Performance of new LLC placement.

LLC Partition Placement: To analyze the impact of LLC partition placement on Omni-Compute, we study a different LLC placement [29] as shown in Figure 3.16(a). Figure 3.16(b) shows the performance of Omni-Compute with the new LLC placement. This LLC placement is easier for physical layout implementation, but due to dimensional routing, it suffers from higher link utilization on the edge links as seen from the performance degradation when compared to our baseline GPU (Section 3.2). On an average, the new placement scheme leads to a performance degradation of 28% compared to the baseline GPU. With Omni-Compute, the overall performance improves by 56% compared to the no-offload execution. Note that, the performance gains achieved by Omni-Compute for this placement are relatively higher than the one achieved for the

baseline GPU. This is due to the fact that more computation offloading can be done in this placement due to the proximity of the LLCs (two LLCs are close to each other) allowing more offload chains to find a suitable EMN.

Shared Memory Optimizations: Applications such as `RED` heavily make use of shared memory in GPUs. This limits the scope of our computation offloading. To this end, we modified the source code of `RED` to make use of global memory rather than shared memory. We also made multiple different variations that use a hybrid approach consisting of global

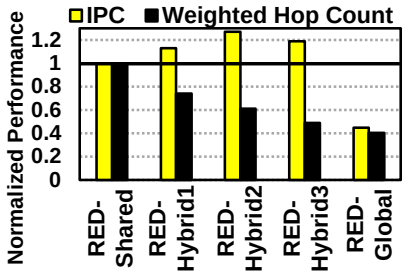


Figure 3.17: Impact of shared memory optimization.

and shared memory. Initial stages of reduction are done using global memory, while the later stages are done using shared memory. Figure 3.17 shows the performance and weighted hop count for five different versions of `RED` using LLC-Compute. The first is the unmodified version that uses only shared memory, while the `RED-Global` performs using global memory. Similarly, three different hybrid approaches (`RED-HybridN`) were prepared where the first N level of reduction happen in global memory and then the following levels use shared memory. `RED-Hybrid2` achieves the best performance, and we use this variant for our experimental analysis in Section 3.6.1.

OQ and SQ Size: To determine the size of OQ and SQ, we performed a sweep of multiple (OQ entries, SQ entries) sizes from (12, 24) to (128, 256) to find a feasible design point. We keep SQ size larger than OQ as each SQ will handle requests from multiple cores whereas each OQ is only used by its core. The performance gains of Omni-Compute plateau at 34% for (64,128) and onwards. This is because most of the offloaded chains can be

accommodated and are not discarded due to OQ/SQ being full. We choose the design point of (48,96) due to the resident hardware wavefront limit of 48 for a core. Specifically, without wavefront/memory divergence, there can be a maximum of 48 offload chains from a given core. The SQ size of 96 was chosen empirically.

Dedicated ALU Units: As mentioned in Section 3.4.3, in Omni-Compute, the ALU is shared between the core and the SQ. We also studied the effects of adding dedicated ALUs of varying SIMD-width towards the average performance improvements (only for the 9 workloads) achieved by Omni-Compute. We observe that, for a 4-wide, 8-wide, and 16-wide ALU, average performance degrades by 12%, 5% and 2%, respectively, over the shared ALU scenario (the LLCs have dedicated 32-wide ALUs). With a 32-wide ALU, the performance improves only by 3% compared to shared ALU.

3.7 Related Work

GPU Optimizations and Computation Offloading: There have been many studies in the past on optimizing the GPU architecture [54, 104, 105, 160, 161, 165–170] and on computation offloading and scheduling [76, 149, 171–175]. Meng *et al.* [169] developed a dynamic warp/wavefront subdivision scheme where the wavefronts are split into smaller units and scheduled at a finer granularity to reduce stalls. Zhang *et al.* [160] proposed multiple heuristics for removing the warp and memory divergence using data reordering and job swapping. These proposals reduce the divergence in an application and can potentially increase opportunities for offloading, therefore, complementing our proposed NDC schemes. Moreover, many prior works [153, 155, 156, 176] have identified on-chip bandwidth in GPUs to be a bottleneck as well.

Near Data Computing (NDC) Architectures: The idea of moving computation closer to data is not new, since it has been studied in different contexts including the memory system, known as PIM [60, 66, 68, 113, 177]. While the PIM concept can be traced back

to early 1970s [65], due to technological limitations, it could not be fully realized. Recent advances in 3D stacking technology have rejuvenated the interest in PIM [30, 40, 63, 71, 122, 125, 171, 175, 178, 179]. Hsieh *et al.* [76] proposed programmer transparent schemes for offloading code segments to PIM cores and co-locating code and data together in a multi-PIM scenario. Tang *et al.* [162] proposed a software approach, which partitions loop statements into sub-statements to reduce data movement in a CMP. Our work is different in two aspects. First, we target a GPU architecture, whose memory access pattern is more complicated due to massive number of parallel threads. Second, their approach requires synchronizations to ensure correctness. Such synchronization is unsafe and very costly in GPUs. Hashemi *et al.* [123] developed a dynamic scheme that migrates computation to memory controllers to reduce the cache miss latency in CMP. While their approach focuses on dependent cache misses to the same memory controller, our approach is more generic and we offload computations to potentially any location including LLCs and other cores. Any off-chip NDC techniques are complementary to our proposal. Compared to prior efforts, we are the first to explore the notion of earliest-meet node in GPUs.

NoC Optimization: Prior works such as [180–182] have proposed active networking, wherein routers have sufficient intelligence to perform simple operations on packets as they flow through them. Network packet inference at the routers has been exploited by prior works such as [183–185] for better congestion management. Kim *et al.* [154] developed a packet coalescing mechanism for GPUs, that reduces the congestion at the MCs, improves performance and reduces data movement. Kim *et al.* [176] provide VC monopolizing and partitioning support for better bandwidth efficiency in GPUs. Bakhoda *et al.* [153] proposed a "checkerboard" mesh for throughput architectures. We use a variation of their proposed LLC placement for our baseline GPU. While such optimizations can help reduce network latency, the network eventually becomes a bottleneck with large problems and datasets. Our proposal can work hand in hand with these techniques for added benefit.

3.8 Chapter Summary

In this work, we present two complementary computation offloading techniques for minimizing on-chip data movement in GPU architectures, and hence, improve performance and energy efficiency. The first technique enables computational offloading to the LLCs, while the second technique complements the first technique by adding offloading capability to any node in the 2D mesh interconnect. We identify several small and basic instruction chains in GPU applications that can be offloaded to any one of the locations. The required compiler support, hardware modification to the cores and LLC are presented to facilitate the NDC mechanisms. Simulation results show that our proposed mechanisms are quite effective for reducing on-chip data movement to improve performance and energy efficiency in modern GPUs.

Chapter 4

Design and Analysis of Control-Flow and Memory Divergence-aware Scheduling in Throughput Processors

In the previous two chapters, we proposed techniques to improve performance and energy efficiency via intelligent computation offloading between GPU and PIM units and computation offloading between different GPU cores and last-level caches, respectively. However, in spite of the many architectural innovations (including the optimizations we propose in this dissertation) in designing state-of-the-art GPUs, their deliverable performance falls far short of the achievable performance due to several issues. One of the major issues that affect GPU performance is due to divergence in control flow and memory requests. Control flow divergences arise due to conditional statements in programs and memory divergences arise due to irregular memory access patterns and both these divergences hurt the warp-level parallelism. These two divergences are specifically not conducive to irregular applications. In this context, this chapter presents a unified software–assisted hardware mechanism, called Shadow Engines (SE), to mitigate the

effects of control-flow and memory divergence.

4.1 Introduction

GPU architectures are highly parallel and have the hardware capability to easily scale and execute thousands of parallel threads at any given instance. Instead of managing each thread separately at both the programming language level as well as the hardware level, the GPU model restricts the execution granularity to a group of threads that execute in lock-step fashion, often referred to as a warp or a wavefront. In lock-step, it follows a single-instruction, multiple-data (SIMD) paradigm, and therefore, it only needs one fetch and decode unit per warp, while the number of execution units (SIMD lanes) are usually kept equal to the number of threads in a warp [27, 186]. This greatly improves the energy-efficiency of GPUs and simplifies the programmability effort. These design decisions work well for general purpose applications that are data parallel with regular memory access patterns and little to no control-flow in their instruction stream (e.g, graphics processing, streaming computations, etc. [187, 188]). Towards addressing the inefficiencies when processing control-heavy, irregular memory access ridden applications, improvements such as better caching strategies, prefetchers, unified virtual memory between CPU and GPUs are being constantly added [189–191]. This has made modern GPU architectures more friendly to various general purpose applications [192–194].

However, as more applications get ported to GPUs, their inherent bottlenecks become more pronounced. Specifically, even after all the improvements with caches and prefetchers, many classes of applications with irregular behavior (both control and memory), heavily under-utilize GPU resources leading to sub-optimal performance [91, 195]. Irregular behavior in an application stems from two different types of divergences, viz. *control-flow divergence* and *memory divergence*. Both divergences arise due to the GPU executing warps in a lock step fashion. Consider a scenario when a warp

executes a conditional instruction, where the result may be different for different threads in the warp, making some threads jump into the *taken* block, while others may go to the *not taken* block. This is called *control-flow divergence* [165] or branch/SIMD divergence, and leads to severe under-utilization of the SIMD lanes, thereby reducing performance. Consider another example where a warp executes a memory request operation. Here, all the threads in the warp generate their respective memory addresses and go through a coalescing stage. This coalescing stage minimizes the number of memory requests by combining memory requests that are mapped to the same cache line. Ideally, when all the threads access contiguously stored data in memory, the individual memory addresses can be coalesced into one cache line. However, in applications with irregular memory access patterns, each thread in a warp can potentially generate memory addresses that are mapped onto different cache lines, requiring multiple memory requests into the memory hierarchy. It could so happen that the different requests are serviced at different times, and there may be threads that already have received their data, yet cannot make forward progress until all the threads in the warp have received all their respective data. This is called *memory divergence* [196], and it also leads to reduced number of *ready* warps that the GPU can context switch into to hide any long latency operations.

To illustrate the impact of control-flow divergence and memory divergence, Figure 4.1 shows the potential speedup that is achievable for 32 applications when they do not incur any divergence during execution (mimicking a multiple-instruction-multiple-data (MIMD) paradigm). We observe that the performance improves by up to 200%, and on an average, by 46% when all the divergences are removed from the application execution. Traditionally, as the warps are executed in lockstep, control-flow divergence is handled by executing both the taken and not taken blocks for all the threads in the warp, but only the threads that are executing their desired block are active while the other threads are marked disabled, hence wasting the SIMD unit resources. On the other hand, memory divergence is handled by stalling the divergent warp until all the memory requests generated by the memory operation are serviced.

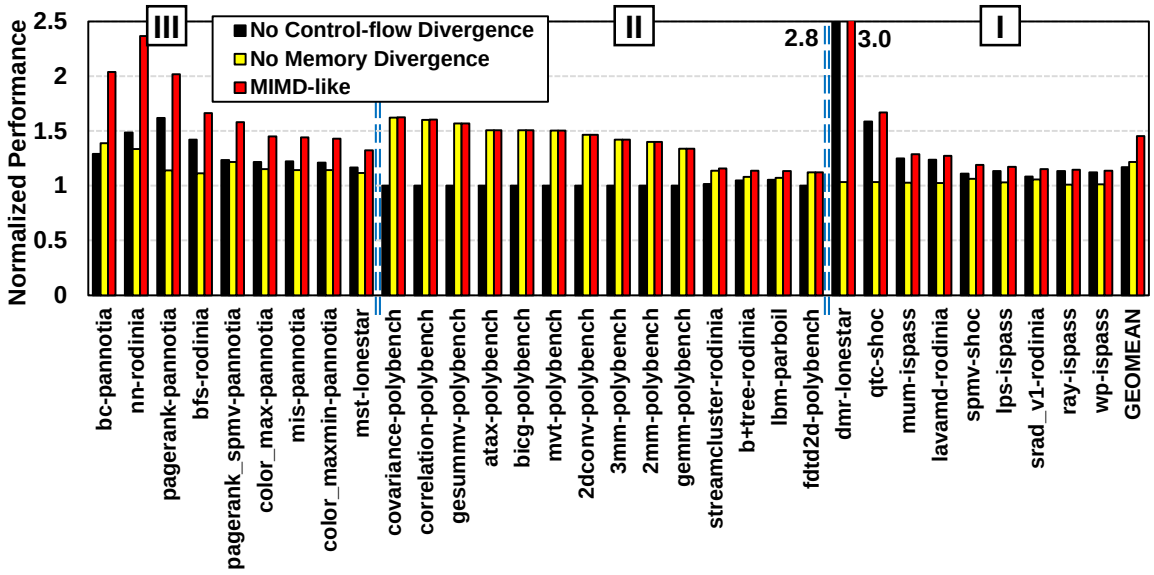


Figure 4.1: Ideal performance achieved without control-flow divergence and memory divergence. The results are normalized to the baseline execution that suffers from divergence.

Considerable efforts in the past have focused on addressing these divergences [160, 161, 165, 169, 197, 198]. For example, Fung *et al.* [161] proposed a thread block compaction mechanism that is able to dynamically generate warps at divergent branches. Meng *et al.* [169] developed a dynamic warp subdivision mechanism that essentially divides a warp into sub-divided warps and schedules these sub-divided warps accordingly. Zhang *et al.* [165] proposed a pure software approach called G-Streamline which uses compiler analysis to reshape the computations to remove control-flow divergence and, at the same time, perform data layout optimizations to reduce the overheads of generating divergent memory operations. Most efforts have addressed the issue of control-flow divergence but do not address the memory divergence challenges. In some cases these solutions may worsen the memory divergence [160, 161, 165]. Moreover prior works that address both control-flow divergence and memory divergence tackle them at compile-time via intelligent code analysis and data layout optimizations [160, 197]. However, they fall short for applications that have data dependent irregularity and the memory access patterns are not known a priori, where invariably the divergence problems are more acute.

It is non-trivial to resolve the control-flow and memory divergence problems in GPUs. This is due to two main reasons. First, simply reducing the SIMD width goes contrary to the motivational reasons for GPUs, where we want very high degrees of parallelism. State-of-the-art GPUs have generally become wider to improve energy-efficiency gains [25, 199, 200] rather than narrower. Second, even though control-flow divergence has been tackled successfully in the past, the dynamically formed warps can lead to increased memory divergence. Similarly, if the warps are generated to optimize for memory divergence, the control flow divergence can worsen. Hence, control-flow and memory divergence need to be tackled in conjunction to improve the deliverable performance of GPUs, specifically with the growing class of irregular applications.

This work proposes a comprehensive technique to minimize the impact of *both* these divergences. We first conduct an in-depth characterization of 32 applications to understand the individual and combined impact of control flow and memory divergence. We then develop a software-assisted hardware approach called *Shadow Engine* (SE) to mitigate the severity of both kinds of divergences in GPU applications by holistically orchestrating the generation and execution of warps. The high level idea behind SE is to *identify the points of divergence as early as possible* during the execution. Prior hardware mechanisms resolved the divergence challenges *reactively*, while SE is able to *proactively* determine the divergence, and hence, takes steps to maximize the benefits when divergence actually takes place. To this end, the proposed SE technique utilizes a compiler pass to identify the conditional and memory instructions and *hoists* their respective backward slices as much as possible while preserving correctness. After the hoisted dependence chain, it inserts one shadow instruction corresponding to the conditional/memory instruction (but well before the real conditional/memory instruction). Note that these shadow instructions are similar to speculative instructions in the sense that they do not change the microarchitectural state of the GPU pipeline, except to be specifically used to populate the SE hardware. Furthermore, by hoisting the dependence chain up, we resolve all the data dependences early, and therefore, the shadow

execution always provides outputs that match the actual output of the corresponding instruction execution. Based on the shadow instructions' information, the SE performs intelligent warp scheduling to maximize the benefits when divergence occurs. For example, let us consider a conditional instruction whose shadow instruction is already executed and SE hardware is already populated. Using the populated information, a SE can decide whether to create new warps or not in order to improve the SIMD utilization. Note that the SE would have *proactively* started prioritizing warps with similar behavior to make forward progress when it encountered the shadow instruction, and therefore, have enough threads available at the point of divergence to regroup. Similarly, when a divergent memory instruction is executed, the SE is notified, which takes over warp scheduling to improve utilization of divergent memory instructions as well. The goal in this work is to develop a mechanism to fully and automatically mitigate the effects of control-flow and memory divergences by generating new warps to reduce control-flow divergence and by performing intelligent warp scheduling to maximize the consumption of divergent memory operations. The main **contributions** of this chapter are the following:

- It provides an in-depth characterization of the control-flow divergence and memory divergence in GPU applications. Our characterization shows that for a hypothetical system (MIMD-like), on an average, we can boost performance by 46% with complete elimination of both divergence types.
- We propose a software-assisted hardware mechanism, Shadow Engine (SE), that identifies divergence points as early as possible during execution to take corrective actions.
- By utilizing the early notifications about the divergence information, we propose two complementary techniques to improve the core utilization and hence performance. For tackling control-flow divergence, we employ a thread regrouping mechanism to minimize stalls and for memory divergence, we employ a locality-aware warp scheduling.

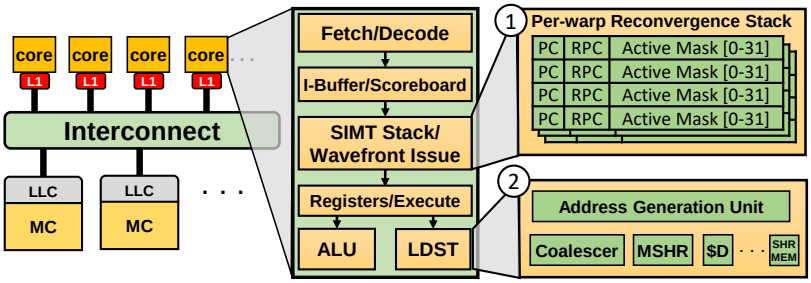


Figure 4.2: Baseline GPU execution pipeline.

- Detailed evaluations of our proposed Shadow Engine mechanism using 32 GPU workloads demonstrate that the combine approach is more effective than individual solutions to either control-flow or memory divergence as has been done in prior work. Moreover, the combined solution provides better performance than simple additive improvements due to each divergence. It provides, on an average, 25.9% improvement in performance (IPC) compared to the baseline architecture, while improving the average core utilization by 14% and lowering coalescing stalls by 24%.

4.2 Background

This section provides an overview of the GPU architecture and the divergence problem.

4.2.1 GPU Architecture

Figure 4.2 shows a simplified view of our baseline GPU architecture. Each core consists of a fetch/decode unit, a scoreboard to check for dependencies, a per-warp SIMT stack ① to keep track of control-flow divergence in each individual hardware warp, a register access unit with operand collector and the execution units (ALU and LDST). The LDST unit ② comprises of the address generation unit (AGU), a coalescer for reducing the number of memory requests generated by a warp, a miss-status holding register (MSHR), a private L1 data cache, a texture cache and a constant cache, along with a software- managed scratchpad

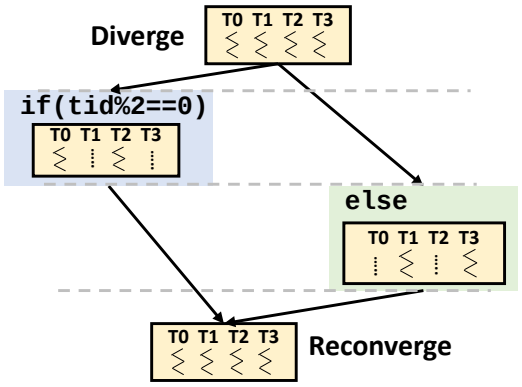
memory (called shared memory). The cores are connected to the memory channels via an interconnect. Each memory channel is directly connected to the L2 cache (LLC) partition that is shared by all the cores, and a memory controller (MC).

To execute a warp, the fetch unit fetches the instruction, decodes it, and stores it in the instruction buffer. This buffer is hard partitioned for each hardware warp. This helps keep the pipeline full so that on a context switch, decoded instructions for the particular warp are already buffered and ready for execution. The decoded instruction is then sent to the scoreboard to check for dependencies. If no dependencies are found, the instruction is then sent to the SIMT stack to account for any control-flow divergence. Once the warp is issued for execution, depending on the instruction type, it is either sent to the ALU or the LDST unit. For memory instructions, the LDST generates the memory addresses for each of the threads in the warp, and then the coalescer reduces the number of memory requests which are then sent to the memory hierarchy for service. If there is more than one memory request that is generated per warp, then the warp is stalled until all of the requests have been serviced.

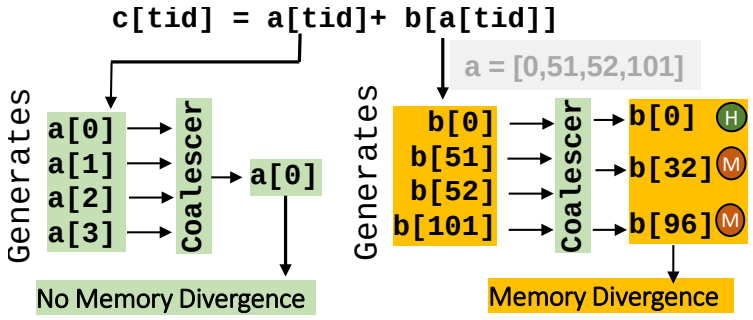
4.2.2 Divergence in GPUs

To illustrate the effects of divergence, let us look at Figure 4.3a. Let us assume that there are 4 threads in a warp (T0—T3), and all the threads are executing the conditional code snippet (`if(tid%2==0)`). Once the conditional instruction is executed by all the threads, the predicate mask is populated as shown in the figure. The `If` block is executed for T0 and T2, while T1 and T3 execute NOPs. Similarly, the `Else` block is executed by T1 and T3, while T0 and T2 execute NOPs. Then, at the reconvergence point[1], all the threads become active. Note that, during the divergent execution, only 50% of the SIMD lanes were active leading to under-utilization. Similarly, for memory divergence let us look at Figure 4.3b.

[1]In our baseline architecture, the reconvergence point is decided based on the immediate post-dominator [201].



(a) Control-flow divergence.



(b) Memory divergence.

Figure 4.3: Illustration of the two types of divergences in a GPU.

Let us consider the execution of the statement `c[tid] = a[tid]+ b[a[tid]]` by the same 4 threads. Here, the instruction is broken down into load/store and ALU instructions. For the 4 threads, the load for `a[tid]` generates 4 memory accesses to the addresses a[0], a[1], a[2], and a[3]. The coalescer combines the addresses and requests only one cacheline, a[0] which contains the other needed data as well[2]. On the other hand, the load for `b[a[tid]]` generates the memory addresses b[0], b[51], b[52] and b[101] based on the values of array `a[]`. The coalescer combines the requests but generates 3 cache line requests corresponding to b[0], b[32] and b[96]. Now, assuming only b[0] *hits* in the L1-cache, thread T0 can potentially make forward progress. However, due to SIMD execution, the warp is still stalled until all the requests (b[32] and b[96]) have been serviced. This leads to memory divergence stalls, which can potentially be exposed at runtime and degrade performance.

[2]One cacheline is 128 bytes long and we assume 4 bytes per data element.

Note that both these divergences are program dependent and thus, cannot be solved solely by architectural optimization. Moreover, the effects of divergences become prominent in irregular applications.

4.3 Motivation

Let us revisit Figure 4.1 to understand the divergences issues in detail to support our design decisions. The figure shows the ideal performance for 32 GPU applications in the absence of any kind of control-flow and memory divergence, essentially executing the application in a MIMD paradigm. We observe that, on an average, performance improves by 46%. Therefore, the SIMD nature of the GPU architecture poses an important bottleneck to address. Furthermore, based on the ideal performance improvements (from Figure 4.1), we classify the evaluated 32 applications into three categories based on the type of divergence they are sensitive to: (Category-I) control-flow divergence; (Category-II) memory divergence; and (Category-III) control-flow and memory divergence.

In the following three sub-sections, we discuss the two divergences and viable approaches to solve the problems.

4.3.1 Analysis of Control-flow Divergence

To analyze the impact of control-flow divergence, we evaluated 32 GPU applications (evaluation details are discussed in Section 4.5) and measured their core (SIMD lane) utilization and details of control-flow divergence across warps. Figure 4.4 plots the average SIMD unit utilization[3] and it can be seen that the performance improvements (from Figure 4.1) can be co-related with their SIMD unit utilization. We also observe that the SIMD utilization and distribution confirm our classification of the applications.

[3]The SIMD units are 32-wide in our baseline architecture. Note that, we do not consider stalls for computing the average SIMD utilization as it will bias the average to the lower-end due to SIMD utilization being 0 when stalled.

In Figure 4.4, a utilization of 32 indicates that all the SIMD lanes are occupied during the entire execution of the application (limited scope for improvement if control-flow is resolved) and a utilization close to 0 indicates that barely any of the SIMD lanes are occupied during execution. We make three observations here. First, there exists applications such as `bc`, `bfs`, `lavamd`, etc. whose average SIMD utilization is lower than 16. Second, applications such as `bicg`, `scan`, `b+tree`, etc. have SIMD utilization very close to the optimal (>30). Finally, there exists applications such as `mis`, `color_max`, `lps`, etc. whose SIMD utilization lie between 16-30. Overall, Category-I and Category-III applications have lower SIMD utilization, while Category-II has high SIMD utilization, and therefore, the performance improvements that can be achieved by eliminating control-flow divergence is lower for Category-III when compared to the other categories (Figure 4.1).

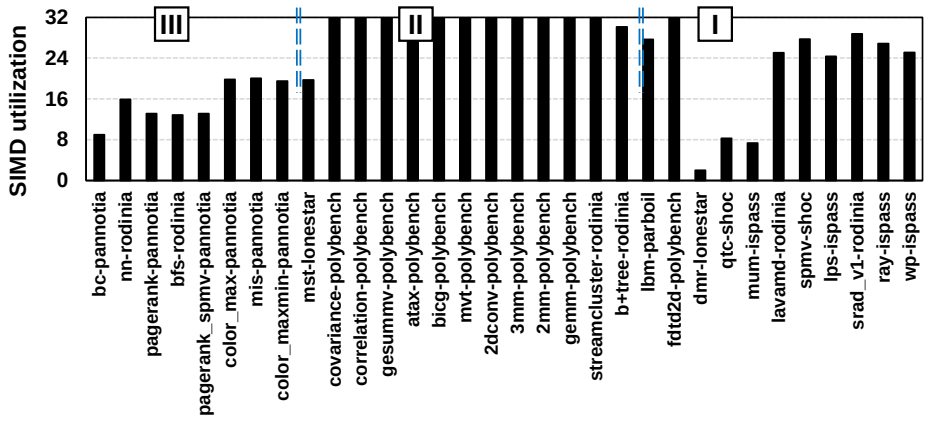


Figure 4.4: Average core (SIMD lanes) utilization.

Figure 4.5 shows the distribution of SIMD unit utilization across 4 bins. We observe that most of the applications have either less than 8 active threads (yellow bars) or more than 24 active threads (pattern bars), while very few applications have warps with active threads between 8-24. This highlights that the control-flow divergence is skewed and is not uniformly distributed. This information is critical to make efficient design choices for our proposed mechanism as discussed in Section 4.4.

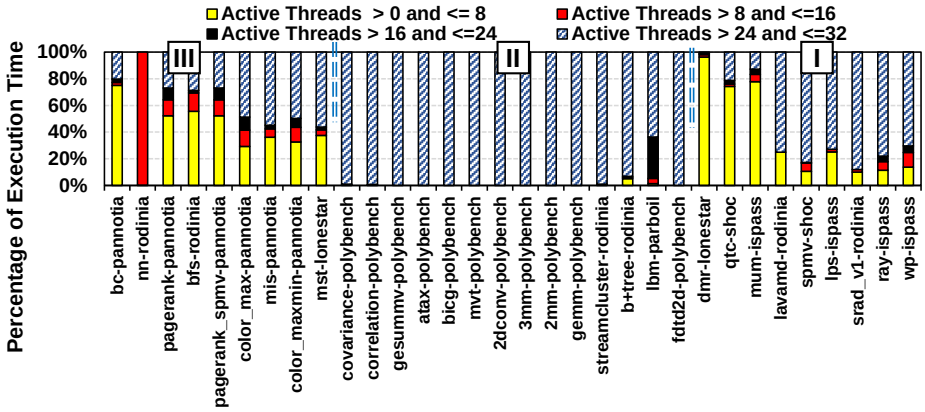


Figure 4.5: Distribution of SIMD lanes utilization.

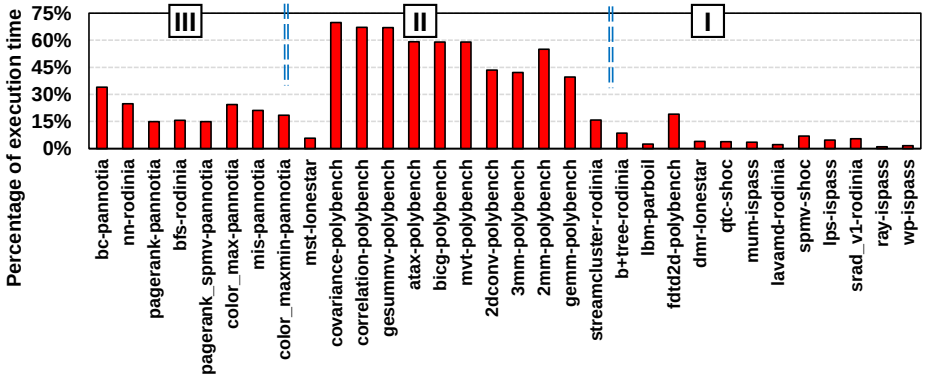


Figure 4.6: Coalescing stalls due to memory divergence.

4.3.2 Analysis of Memory Divergence

To analyze the impact of memory divergence on performance, we eliminate the stalls that were introduced due to it. Figure 4.1 ("No Memory Divergence") shows this performance improvement. Note that, control-flow divergence can still persist in this scenario. On an average, the performance improves by 21.5%. This increase in performance is due to the forward progress that is made by threads in a warp that have received their data, which otherwise would have been stalled. Figure 4.6 shows the coalescing stalls that are incurred by an application as a fraction of the total execution time. Coalescing stalls are defined as the number of cycles a warp is stalled while it is waiting for *some* of its data to be received. We make three observations from this plot. First, applications such as `gaussian`, `lps`, etc.

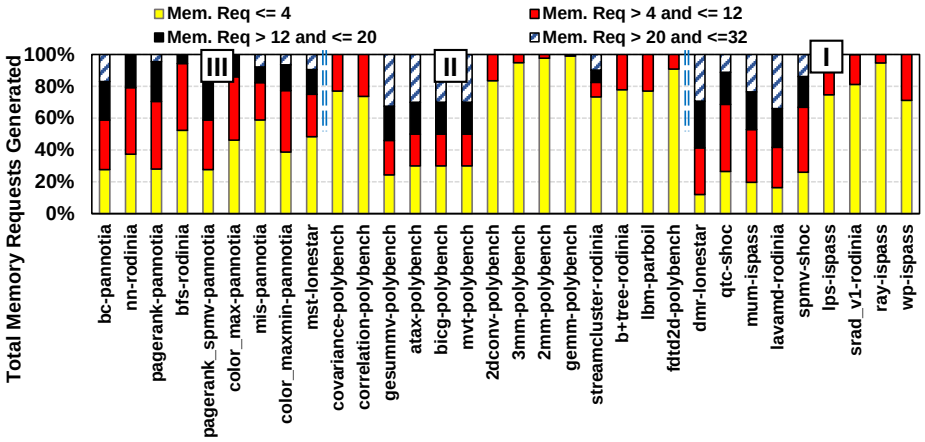


Figure 4.7: Distribution of memory requests generated per memory instruction.

have coalescing stalls less than 10% of the execution time. Second, applications such as `bc`, `mis`, etc. have coalescing stalls less than 35% of the execution time. Finally, there are applications such as `bicg`, `covariance`, etc. that have more than 40% of their execution time being spent as coalescing stalls. Note that coalescing stalls need not be completely exposed at runtime. It can be hidden due to the SIMT nature of the GPU architecture. Still, higher the coalescing stalls, less likely of it being hidden as the number of ready warps to context switch with also reduce greatly. Overall, Category-III and Category-II applications have medium to high coalescing stalls, while Category-I has low coalescing stalls. Therefore, the likelihood of improving the performance by eliminating memory divergence is lower for Category-I applications when compared to the other categories (Figure 4.1).

We also analyze the number of memory requests that are generated by all the memory instructions of an application and categorize them into four different bins depending on their intensity of number of memory requests generated (shown in Figure 4.7). We observe that most of the applications generate less than 12 memory requests per memory operation, while a small subset of applications such as `mst`, `bicg`, etc. generate more than 20 requests per memory instruction.

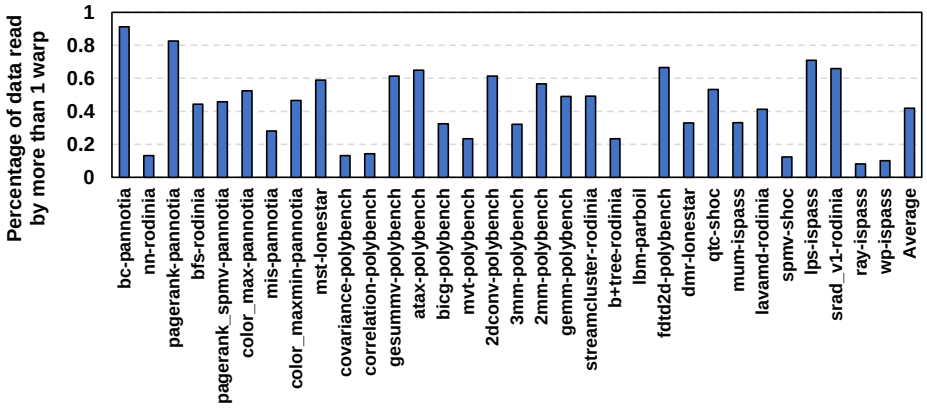
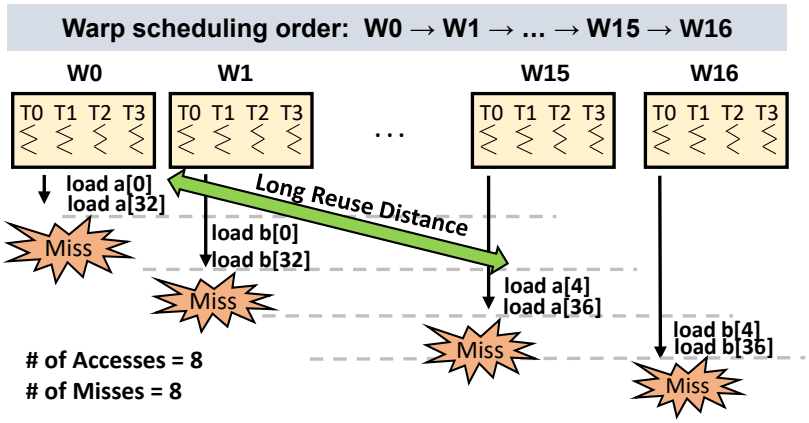


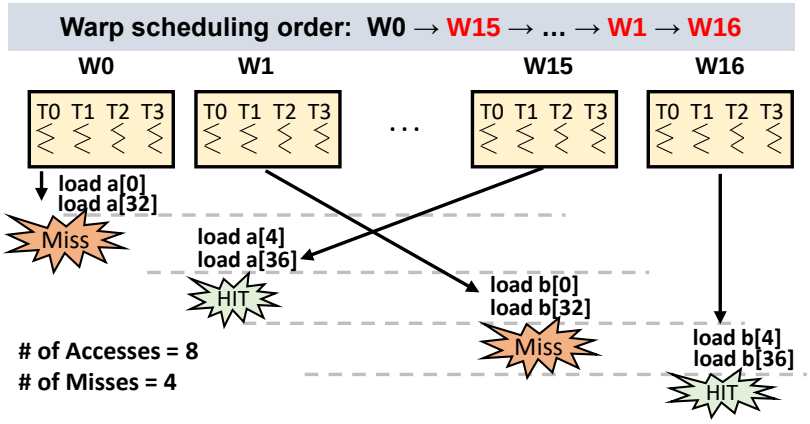
Figure 4.8: Percentage of data accessed by more than one warp.

4.3.3 How to Reduce Divergence?

In order to mitigate the effects of control-flow divergence, the SIMD unit utilization needs to be increased – preferably, it needs to be equal to the SIMD unit width. To this end, threads need to be remapped to form groups that follow the same path during the divergent parts of the code. Tackling memory divergence is non-trivial as there are many potential ways of resolving it. One potential approach is to perform data layout transformation to achieve an appropriate thread-data mapping. However, this has high data movement and bookkeeping overheads. Rather, one can potentially schedule warps intelligently to maximize the data consumption of the divergent memory operations. Figure 4.8 shows the percentage of sharing that occurs across warps over their total data accesses. We observe that, on an average, 41% of the total data is accessed by more than one warp, and therefore, intelligent warp scheduling can help achieve performance improvement. Let us consider the two scenarios shown in Figure 4.9. Scenario I: when 4 warps generate 2 load requests each for a total of 8 load requests but effectively only accessing 4 cachelines. However, the long reuse distance due to the warp scheduling policy, by the time the cachelines are reused by W15 and W16, the data has been evicted out and it results in a cache miss. Scenario II: In this case, after scheduling W0, W15 is scheduled, which translates to a *hit* (either a cache hit or an MSHR hit). Similar will be the case when W16 is scheduled immediately after



(a) Scenario I: Agnostic to memory divergence.



(b) Scenario II: Aware of memory divergence.

Figure 4.9: Different warp scheduling scenarios.

W1. Note that, although the coalescing stalls do still exist for W0 and W15, a portion of it has been overlapped due to the intelligent scheduling, and furthermore, the W15 misses have been converted to *hits*, leading to an efficient execution. Moreover, even if the data layout is optimized and each warp generated 1 load request, it would still require 4 load requests (as there are 4 distinct cachelines), which is also the case for Scenario II that makes use of memory divergence aware scheduling. Therefore, in this work, we limit ourselves to intelligent warp scheduling and do not explore any potential data layout optimizations, as discussed in Section 4.4.4.

4.4 Design of Shadow Engine

The main idea of the **Shadow Engine (SE)** is to dynamically mitigate the effects of any control-flow and memory divergence that arise during run-time. To this end, we need to essentially perform the following two steps: (1) dynamically form new warps that maximize the SIMD lane utilization, and (2) improve the consumption of the multiple memory requests generated from a single warp through intelligent warp scheduling in order to hide any exposed memory divergence.

4.4.1 Design Challenges

There are two main challenges that need to be tackled in order to build SE. First, the reshuffling of threads to create new warps *requires the knowledge of the paths to be taken* by each of the threads and the availability of such threads that are executing the same

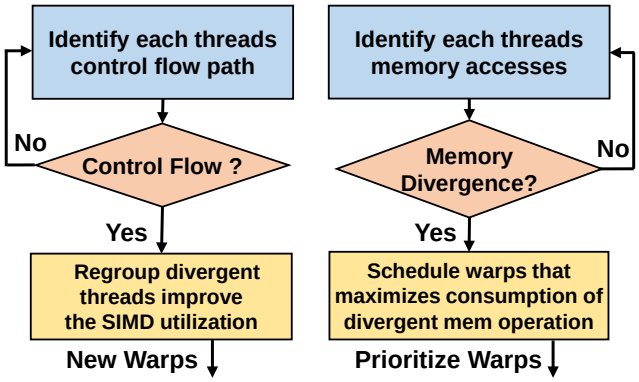


Figure 4.10: Key steps to mitigate divergences.

instruction. Second, to efficiently mitigate the effects of the latency exposed due to memory divergence, we need to be able to identify *which warps should be scheduled close in time* in order to maximize the consumption of the divergent memory instructions.

Figure 4.10 shows the steps in facilitating our proposed mechanism: (1) identify the control-flow paths for each thread; (2) identify the divergent memory operations for each

warp; (3) in the presence of control-flow divergence, re-group the threads to form warps with higher SIMD lane utilization; and (4) in the presence of memory divergence, modify the warp scheduling to prioritize warps that will maximize the consumption of the divergent memory operations.

4.4.2 Proposed Mechanism

As mentioned earlier, there is a need for a priori knowledge of the threads control-flow path and the memory addresses it accesses. Furthermore, we need hardware support to facilitate regrouping of threads, tracking memory divergence, and intelligent scheduling of warps.

Identifying Control-flow and Memory Divergence: In order to ensure that we can safely predict the control-flow path and memory divergence, we use a code hoisting mechanism [202, 203]. Note that, we perform all code analysis on the Parallel Thread Execution (PTX) ISA generated by the CUDA compiler [204]. Figure 4.11 shows the code hoisting mechanism used to identify the control-flow path of each individual threads. The entire control-flow instruction (`setp`) and its dependent instruction chain are hoisted to the earliest point possible in the instruction stream. Note that the hoisting does not cross basic block boundaries. Furthermore, the `setp` instruction is replicated (*as a shadow instruction*) unlike its dependent chain which is hoisted up. We develop these shadow instructions such that they do not modify any architectural registers and only interact with the Shadow Engine (SE) (described below) to provide the required control-flow information. The real `setp` instruction remains in its original position and acts as a divergent point.

Similarly, Figure 4.12 shows the code hoisting mechanism for memory divergence mitigation. Each of the memory instruction's corresponding shadow instruction is inserted and hoisted up along with the dependent chain for the memory instruction's effective address generation. These shadow instructions also interact directly with the SE providing

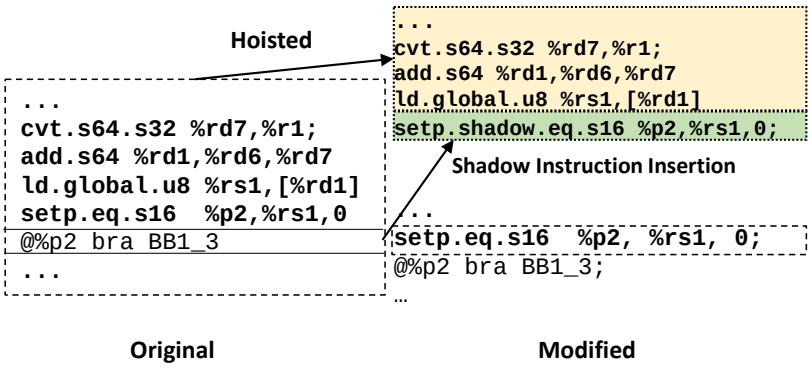


Figure 4.11: Code hoisting for conditional instructions.

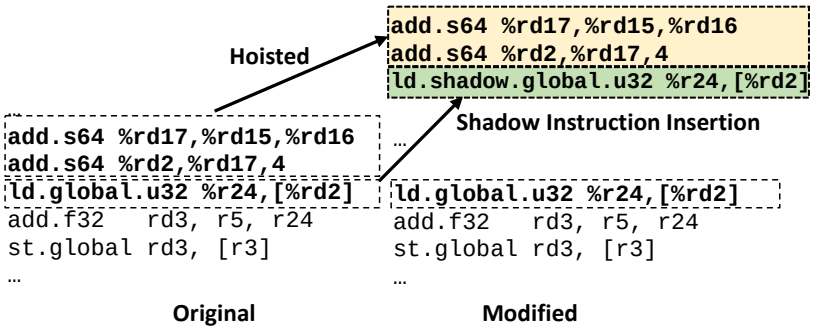


Figure 4.12: Code hoisting for memory instructions.

with the information of the memory address the thread will access. Note that shadow instructions for memory operations do not perform any memory loads/stores. They are executed only to generate the coalesced memory addresses for each warp and perform L1 cache probing on the generated requests to confirm the presence of memory divergence. Note that, we perform code hoisting to help SE direct the execution schedule of the warps in order to perform a control-flow and memory divergent-aware scheduling.

Hardware Support for Shadow Engine: Figure 4.13 shows the hardware modifications that are needed to the baseline architecture to mitigate control-flow and memory divergence. Specifically, we add an additional component called Shadow Engine (SE) (❶) to the GPU pipeline. SE is tightly integrated with the SIMT Stack and the Warp Issue Unit, and is responsible for keeping track of control flow divergence and memory divergence, warp shuffling and warp prioritization. SE is made up of six components: Shadow Controller

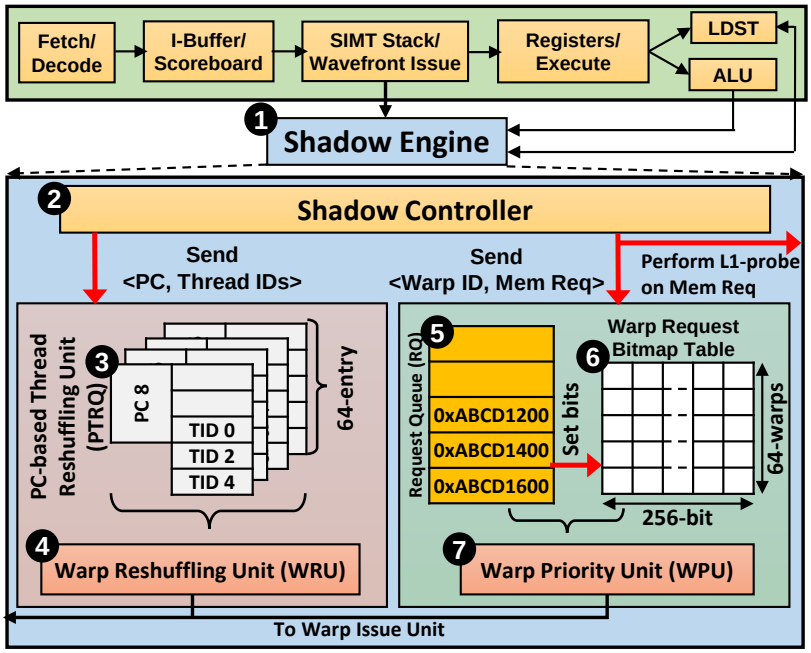


Figure 4.13: Proposed hardware design of Shadow Engine.

(❷), PC-based Thread Reshuffling Queues (PTRQ) (❸), Warp Reshuffling Unit (WRU) (❹), Request Queue (RQ) (❺), Warp-Request Bitmap Table (WRBT) (❻) and Warp Priority Unit (WPU) (❼). The Shadow controller is responsible for receiving the information from the shadow instructions. Based on the shadow instruction type (conditional or memory), the shadow controller accordingly populates an entry either in the PTRQ or the RQ and WRBT. PTRQ is made up of 16 queues of 64-entries each. Each queue is tagged with a Program Counter value and holds threads that are going to execute the tagged PC at the control-flow divergent point. RQ keeps track of the cacheline addresses that belong to divergent memory instructions. The number of entries in RQ is equal to the number of entries in the L1 cache miss status holding register (MSHR) (In our baseline architecture, we have a 256-entry MSHR.). Only entries that have been deemed as L1-miss and contribute to memory divergence are added into the queue. WRBT keeps track of the warps that are going to access the memory address in RQ. It is a 64-entry, 256-wide bitmap, where each entry corresponds to a hardware warp and the bitmap corresponds to the entries in RQ as to whether that address is being accessed by the warp or not. WRU is responsible for

taking 32 entries from a queue of PTRQ and forming new warps dynamically and issuing them, while WPU is responsible for prioritizing certain warps that are likely to access data from divergent memory loads. We give detailed explanation of how these components are utilized in Section 4.4.3.

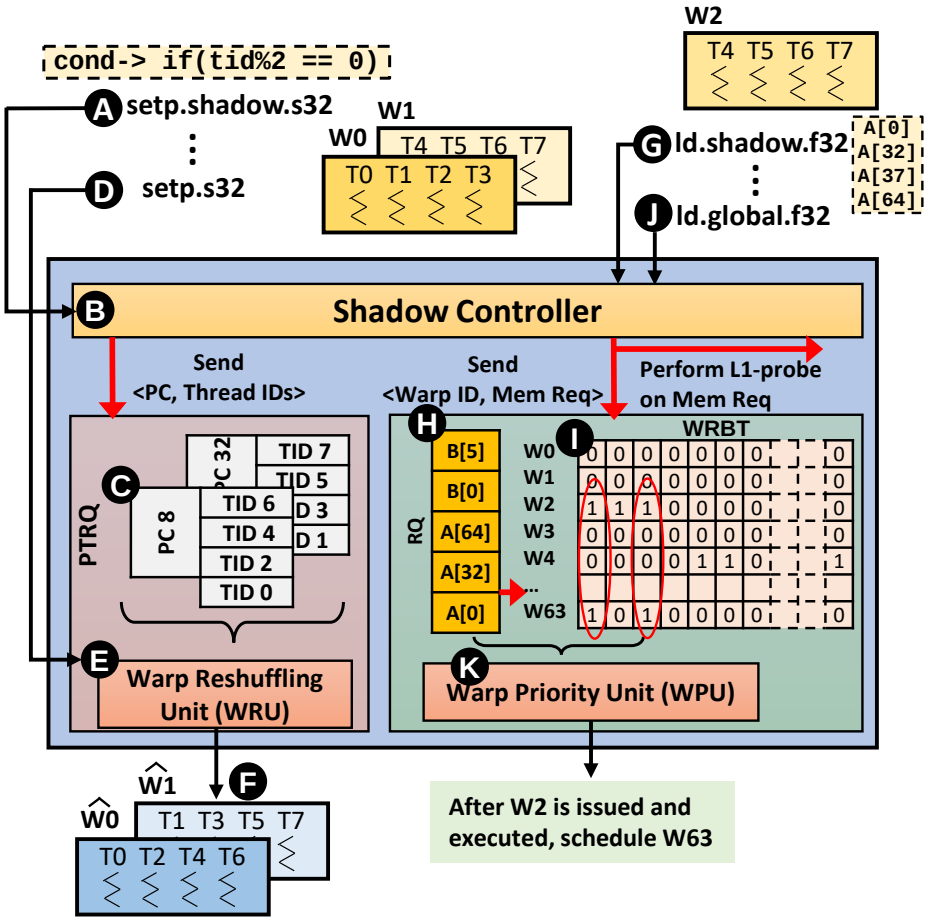


Figure 4.14: Representative scenarios for control-flow and memory divergence.

4.4.3 How does Shadow Engine Work?

In this section, we describe how our Shadow Engine mitigates control-flow divergence and memory divergence.

Control-flow Divergence: Let us refer to Figure 4.14 where Warp 0 (W0) and Warp

1 (W1) have a conditional statement `if(tid%2==0)` to execute. We first perform code hoisting and add shadow conditional statements. When the shadow instruction is executed (Ⓐ) by W0, the predicate bits for the threads in the warp are obtained by the Shadow Controller (Ⓑ), which then divides the threads into two different PC queues (`taken` and `not taken` blocks) and pushes them into the PTRQ (Ⓒ). If there are queues already tagged with the PC for `taken` and `not taken`, the thread IDs for W0 are added to these queues, otherwise a new queue is obtained from PTRQ and tagged with the appropriate PC values. Also, when a queue is tagged with a PC, SE starts prioritizing other warps to fill the queue (going against the baseline warp scheduler (GTO)) until it is filled with at least 32 entries at which point the scheduling reverts back to GTO. Note that, the real conditional statement is a divergent point. When it is executed (Ⓓ), SE is notified, and 32 entries from the queue following the divergent statement as the tag are pulled and used to create a new warp by the WRU (Ⓔ). The new warp ($\widehat{W0}$, Ⓕ) is executed until the reconvergence point. At the reconvergence point, $\widehat{W0}$ is dissolved and the threads go back to their original warps. Note that, when a warp dissolves, its threads can potentially belong to 32 different warps ($\widehat{W0}$ contains threads from W0 and W1). Until all the threads from the original grouping are not dissolved, the original warp continues stalling, while other newly-generated warps ($\widehat{W1}$) execute the divergent instructions.

Memory Divergence: Let us again refer to Figure 4.14 to explain the working of SE during memory divergence. Let us assume that the Warp 2 (W2) is executing a load operation. Similar to control-flow divergence, we hoist the dependent chain of the load operation and replicate a shadow instruction for the load instruction. When the shadow instruction is executed by W2 Ⓖ, the shadow load goes through the load/store unit and generates all the memory addresses and coalesces them to minimize the number of memory requests. If the number of memory requests after coalescing is one, then there is no memory divergence and the execution continues normally. If there are more than one memory requests generated after coalescing, the memory addresses are probed in the L1-cache. If all of the requests are cache hits, again there is no memory divergence and the baseline execution resumes.

Only if there are some cache hits and some cache misses or all cache misses[4], does memory divergence exist. In the presence of memory divergence, all the memory requests (that were cache miss during the L1 probe) after coalescing stage are pushed into the RQ (H) and the corresponding bits (depending on where it was pushed in the RQ) for <W2,RQ entry#> in the WRBT are set (I). When the real load instruction is executed by W2 (J), SE finds the most suitable warp to schedule next based on the entries in WRBT. The warp with the most matching bits with W2 is chosen (in this case W64) and is scheduled to be executed immediately after W2, leading to faster data consumption of the divergent load. In the presence of both control-flow divergence and memory divergence, we prioritize control-flow divergence (prioritize warps to fill PTRQ) and then perform scheduling to optimize memory divergence.

4.4.4 Limitations of Shadow Engine

Control-flow Divergence: We do not consider threads from different thread blocks for formation of new warps due to the existence of synchronization primitives. It is possible to relax the constraints to further improve the SIMD utilization, but we do not explore such optimizations in this work. Also, we use the immediate post-dominator reconvergence mechanism where as there are prior work [165] that make use of likely-convergence mechanism to improve performance further. It is possible for us to make use of this mechanism as well, but we leave it for potential future work.

Memory Divergence: In this work, we only consider intelligent warp scheduling when dealing with memory divergence. We do not consider other approaches such as data layout optimizations. Note that data layout optimizations are not always feasible, since they require fine-grain management of data copy/transfer and computation overlap. If there is no overlap, the performance benefits are not clear and might lead to degradation as

[4]Memory requests that are heading to L2 cache can also return at different times depending on their L2 partition location, bank, etc., and therefore, for any memory instruction that generates multiple requests that are L1 misses will lead to memory divergence.

Table 4.1: Configuration parameters of the GPU.

GPU Features	1.2GHz, 80 cores, 32 SIMT width, GTO warp scheduler
Resources/Core	96KB shared memory, 64KB register file, maximum resident 2048 threads (64 warps, 32 threads/warp)
Private Caches/Core	32KB L1 D-cache, 64KB C-cache, 128KB I-cache, 128B block size
L2 Cache	32-set, 24-way, 128 KB cacheline size L2 cache partition, 192KB/Memory Partition, 4.5MB cache size
Memory Model	24 Memory Controllers, FR-FCFS, 16 banks/MC, 850 MHz, 768 GB/s peak BW
HBM Timing [206]	$t_{CL} = 12$, $t_{RP} = 12$, $t_{RC} = 40$, $t_{RAS} = 28$, $t_{CCD} = 2$ $t_{RCD} = 12$, $t_{RRD} = 3$, $t_{CDLR} = 3$, $t_{WR} = 10$
Interconnect	Crossbar, 1200MHz, dest. tag routing, 2 cores/node, 1VC, flit size=40B, Buffers/VC=256, islip VC & switch allocators

well. Furthermore, data layout optimizations are not always possible for applications with data-dependent irregular access patterns, and are usually feasible only for applications with easy-to-analyze memory access patterns. Finally, we do not modify the baseline thread block scheduling policy (round-robin), which can also efficiently hide the memory divergence latency via intelligent thread block scheduling [168, 205].

4.5 Experimental Methodology

Simulated System: We simulate the baseline architecture as mentioned in Table 4.1 using GPGPU-Sim v4.0 [103]. The code was hoisted at the PTX level and the support for shadow instructions was also added in the PTX ISA. We modeled the appropriate latency for each of the shadow instructions and the additional execution overhead was taken into consideration during simulation. We extensively modified GPGPU-Sim to implement the SE and its related components. SE was added into the GPU pipeline as a shadow to the SIMT-Stack/Issue unit. It is possible to merge the SIMT-Stack and the SE, but to keep the implementation simple, we kept the units different. We run the applications until completion or 1 billion instructions, whichever comes first.

Table 4.2: List of evaluated benchmarks. Legend: (A) Type of divergence sensitivity (I: Control-flow divergence, II: Memory divergence and III: Control-flow and memory divergence), (B): Number of inputs evaluated.

Workload	A	B	Suite	Workload	A	B	Suite
qtc	I	1	shoc [87]	3mm	II	1	polybench [93]
spmv	I	1	shoc [87]	atax	II	1	polybench [93]
lps	I	1	gpgpu-sim [103, 207]	bicg	II	1	polybench [93]
mum	I	1	gpgpu-sim [103, 208]	corr	II	1	polybench [93]
ray	I	1	cuda [92, 103]	covariance	II	1	polybench [93]
wp	I	1	gpgpu-sim [103, 209]	fdtd2d	II	1	polybench [93]
dmr	I	1	lonestar [91]	gemm	II	1	polybench [93]
mst	III	3	lonestar [91]	gesummv	II	1	polybench [93]
bc	III	2	pannotia [210]	mvt	II	1	polybench [93]
color_max	III	2	pannotia [210]	2mm	II	1	polybench [93]
color_maxmin	III	2	pannotia [210]	b+tree	II	1	rodinia [94]
mis	III	2	pannotia [210]	bfs	III	3	rodinia [94]
pagerank	III	1	pannotia [210]	lavamd	I	1	rodinia [94]
pagerank_spmv	III	1	pannotia [210]	nn	III	1	rodinia [94]
lbm	II	1	parboil [211]	srad_v1	I	1	rodinia [94]
2dconv	II	1	polybench [93]	streamcluster	II	3	rodinia [94]

Benchmarks: We analyzed over 140 GPGPU workloads from GPGPU-Sim [103], LonestarGPU [91], Pannotia [210], Parboil [211], Polybench [93], Rodinia [94] and SHOC [87]. However, we only report the analysis for 32 applications and omit the others due to two reasons: (1) There are multiple implementations of the same application (such as `bfs`) across different benchmarks suites that have similar behavior, and (2) many of the applications did not show any performance benefit (<10%) when simulating a MIMD-like system, meaning that they did not suffer from any divergences to begin with. Table 4.2 shows the list of the evaluated benchmarks along with the number of different inputs that were simulated. It also shows the classification category of the application being sensitive to which type of divergence. Note that, for each application with multiple inputs, the average for all the application is presented. Out of 32 GPU workloads, 9 applications belong to Category-I (sensitive to control-flow divergence), 14 applications belong to Category-II (sensitive to memory divergence) and 9 applications belong to Category-III (sensitive to both, control-flow and memory divergence).

Hardware Overheads: In SE, the largest components are the three storage structures: (1) PTRQ, (2) RQ, and (3) WRBT. PTRQ is a group of 16 FIFOs of 64-entry each, with each entry occupying 11 bits for the thread IDs. Therefore, it requires a total of 1.4KB of storage. Note that, we limited the number of queues to 16 in order to handle a maximum of 16 different Program Counters (PCs) values. In the extreme case, this could be very high depending on the number of nested conditions and the discrepancy between the forward progress of different warps. However, in practice we find that more than 16 PCs do not improve performance. This is due to our early identification of divergence points and preparing the warp scheduling to have enough threads with the same PC to be used to form new warps. RQ is a 256 entry storage buffer to hold the memory addresses generated by the shadow instructions. The memory addresses can be of length 64 bits. Therefore, a total of 2KB storage is required for RQ. Note that, RQ can be combined with the L1-MSHR for efficient implementation. WRBT is a 64-entry 256-bit bitmap table that requires a total of 2KBs of storage. Overall, SE requires a storage overhead (per core) of approximately 5.4KBs which is less than 2% of the total storage available in a core of a state-of-the-art GPU architecture. For control units such SC, WRU and WPU, the implementation overheads are negligible.

4.6 Experimental Results

To evaluate the benefits of our proposed Shadow Engine, we measure the GPU performance (instructions-per-cycle [IPC]), increase in SIMD utilization, and reduction in coalescing stalls. The increase in the SIMD utilization highlights better usage of the SIMD lanes in the GPU core. On the other hand, a decrease in coalescing stalls indicates that the warps are waiting less for divergent memory operations to complete.

We compare three different implementations of Shadow Engine:(1) *SE-Control-flow*; (2) *SE-Memory*; and (3) *SE-Both*. *SE-Control-flow* only mitigates control-flow divergence

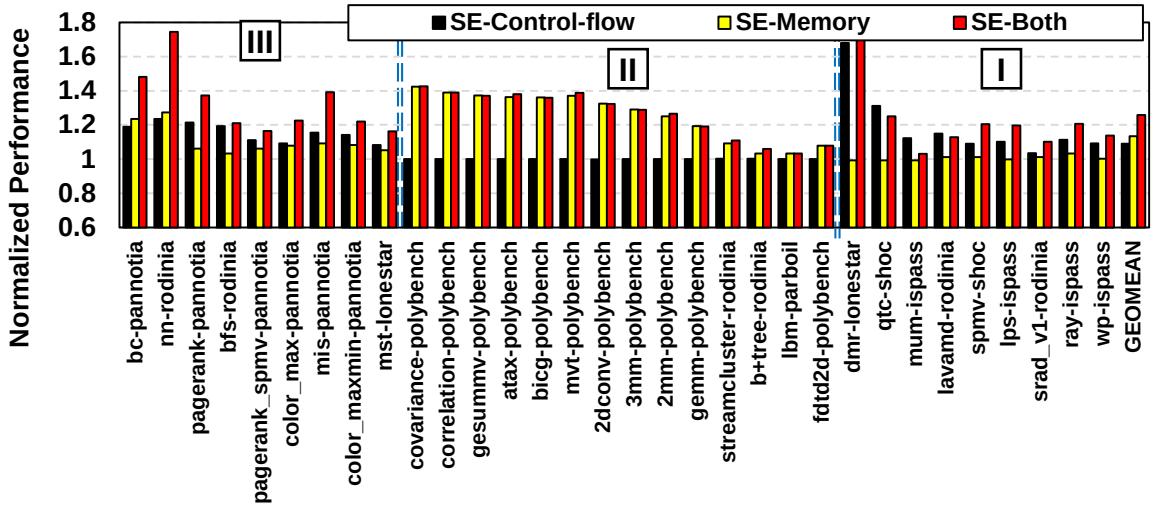


Figure 4.15: Impact of Shadow Engines on performance.

and does not perform code hoisting and shadow instructions for memory instructions. Furthermore, the RQ, WRBT and WPU components are disabled in the Shadow Engine. *SE-Memory* mitigates only the memory divergence through intelligent warp scheduling. Similar to *SE-Control-flow*, it does not perform code hoisting and shadow instructions for conditional instructions, and the PTRQ and WRU components are disabled in Shadow Engine. *SE-Both* mitigates both control-flow divergence and memory divergence. All results are normalized to the execution of the vanilla workloads (without shadow instructions and code hoisting) on the baseline architecture.

Figure 4.15 shows the performance benefits for the three different implementations of SE. For the evaluated 32 GPU applications, *SE-Control-flow*, *SE-Memory* and *SE-Both* improve performance by up to 68%, 42%, 74%, respectively, and on an average, by 9%, 13.5% and 25.9%, respectively.

Effects on Category-I applications: As Category-I applications are sensitive to control-flow divergence, *SE-Control-flow* improves the performance, on an average, by 17.5% while *SE-Memory* does not affect the performance. With *SE-Both*, the average performance improves by 20.7%. *SE-Both* performs better than SE-Control-flow, because once new warps are created dynamically, it leads to memory divergence. As *SE-Both* is able to

resolve memory divergence as well, it provides better improvement than *SE-Control-flow*.

Effects on Category-II applications: On the other hand, Category-II applications are sensitive to memory divergence, and therefore, *SE-Memory* improves the performance, on an average, by 24.7% while *SE-Control-flow* had negligible impact towards performance. With *SE-Both*, the average performance improves by 25%. The slight change in performance is due to applications such as `b+tree` exhibiting low levels of control-flow divergence, which gets resolved in *SE-Both*.

Effects on Category-III applications: As Category-III applications are sensitive to both control-flow and memory divergence, *SE-Control-flow* improves the performance, on an average, by 15.5% while *SE-Memory* improves the performance by 10.5%. As *SE-Both* targets both types of divergences, the average performance improves by 31.9%. Note that the performance achieved by *SE-Both* is larger than the combined linear performance gains for *SE-Memory* and *SE-Control-flow*. This is due to the constructive inference of the two optimizations. Once control-flow divergence is resolved, it leads to an alternate memory access pattern that allows more opportunities for optimizations (memory divergence removal) to take place. Applications such as `bc`, `nn`, `pagerank` and `mis` show this behavior prominently, while applications such as `bfs`, `mst` and `pagerank_spmv` show performance gains that are near to their linear combination of performance gains achieved by *SE-Control-flow* and *SE-Memory*.

To further understand the reasons for the performance improvement achieved due to the mitigation of control-flow divergence, in Figure 4.16, we plot the normalized SIMD unit utilization for *SE-Both*. On an average, SIMD utilization improves by 14%. For Category-I applications, the SIMD lane utilization improves by 27.4% and for Category-III applications, the performance improves by 25.9%. On the other hand, Category-II applications do not see any noticeable change to their SIMD utilization. The SIMD unit utilization correlates well with the performance improvement that are achieved when control-flow divergence is eliminated.

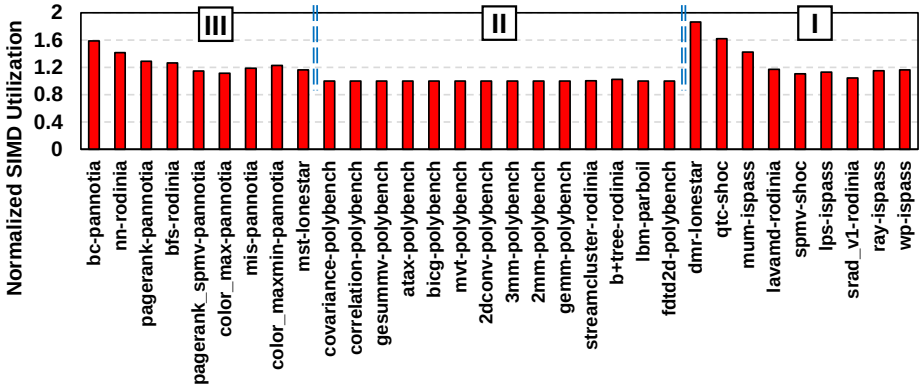


Figure 4.16: Normalized SIMD lanes utilization.

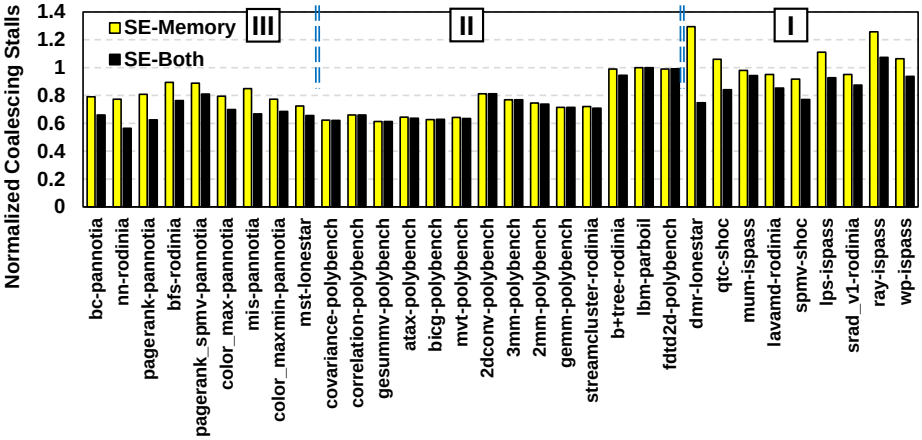


Figure 4.17: Normalized coalescing stalls.

Similarly, to understand the underlying reason for performance gain that is obtained by reducing the effects of memory divergence via intelligent warp scheduling, in Figure 4.17, we show the normalized coalescing stalls for *SE-Memory* and *SE-Both*. On an average, the coalescing stalls reduce by 16% and 24% for *SE-Memory* and *SE-Both*, respectively. This reduction in coalescing stalls can be co-related to the performance gains that are achieved when utilizing a memory divergence-aware warp scheduling. Specifically, the warps are scheduled such that even in the presence of divergent memory operations, the data locality across warps is exploited. Note that, the coalescing stalls may only be partially exposed to the run-time and therefore, any reduction in the stalls will manifest differently for different applications based on their latency hiding capability. This is observed for applications such

as `2dconv` and `3mm` where the coalescing stalls do not reduce as much, but the exposed latency reduces due to improved warp scheduling, and hence improves performance greatly. Note that, for certain applications such as `dmr`, `lps` and `ray`, the coalescing stalls have increased. This is due to two reasons: (1) their coalescing stalls are a very small fraction of the execution time (from Figure 4.6), and therefore, the increase is negligible; and (2) these are control-flow divergence sensitive applications and modifying the warp scheduler to improve memory divergence changes the memory access pattern from the baseline execution, which in turn introduces slight inefficiencies.

4.7 Related Work

To the best of our knowledge, this is the first work that comprehensively tackles control-flow and memory divergence concurrently without any data layout optimizations for both regular and irregular applications. Below, we briefly discuss the prior research in the related areas.

Divergence Optimizations: Control-flow divergence has been well studied in the past and many prior works have mitigated it through variety of techniques [161, 165, 170, 198, 212–217]. Few of the prior works also considered the effects of memory divergence along with control-flow divergence [160, 169, 197, 218]. Fung *et al.* [161] developed a dynamic wavefront formation (DWF) technique that dynamically regroups threads into new warps to reduce control-flow divergence. Fung *et al.* [165] proposed a thread block compaction (TBC) technique, which improves upon the dynamic wavefront formation technique by pushing the reconvergence point further than what is allowed in post-dominator based reconvergence mechanism. Unlike our Shadow Engine, neither DWF nor TBC resolves the issues of memory divergence and may in fact increase its severity. Meng *et al.* [169] proposed a dynamic warp subdivision (DWS) mechanism that essentially breaks a warp into smaller warp-splits which then can be scheduled. This effectively reduces the SIMD width

of the warp-splits, and thereby, reduces the effects of divergence. However, when executing a warp-split, there are still idle resources that are not being utilized. Shadow Engines dynamically forms new warps, and therefore, maximizes the resource utilization. To resolve memory divergence, Shadow Engine opts for an efficient warp scheduling mechanism over DWS, which could potentially lead to long reuse distances. Zhang *et al.* [160] developed G-Streamline, a software framework that can dynamically handle both types of divergences by performing thread regrouping and data remapping. They overlap the latency of data remapping with computation. Although, they resolve the issues of divergence in software, unlike Shadow Engines, G-Streamline is unable to capture the control-flow and memory divergence for applications with data-dependent control-flow and irregular memory access patterns.

GPU Optimizations: There have been many prior works on improving the performance of GPU architectures [59,85,105,106,168,196,202,205,219–221]. Rogers *et al.* [105] proposed a divergence-aware warp scheduler that predicts the L1 cache usage of the active threads and schedules warps in order to minimize cache thrashing and increase data locality. Tang *et al.* [168] developed a locality-aware scheduler that takes into account the locality between kernels, thread blocks and warps, and schedules them accordingly at each level of scheduling hierarchy. Lee *et al.* [219] proposed a criticality-aware warp scheduler that categorizes the warps based on their execution latencies, and prioritizes them to improve performance. Rogers *et al.* [222] developed a variable warp size architecture for GPUs where they start off with a smaller warp size and then gang up multiple of these smaller warps execution whenever possible to improve performance and energy efficiency. Lee *et al.* [221] proposed a pattern-based warp scheduler that observes the instruction issue pattern of workloads and dynamically adapts the scheduling strategy to improve performance.

Early Execution/Code Analysis: Many prior works have dealt with code hoisting or speculative execution to improve performance [223–229]. Kim *et al.* [223] developed a warp pre-execution mechanism that identifies independent instructions of a stalled warp

and executes them as long as they are not part of a long-latency dependence chain. This increases the latency hiding capability as well as register utilization. Menon *et al.* [226] proposed iGPU, a mechanism which adds support for exception handling and speculative execution in GPUs. They convert the code into smaller regions which are idempotent to allow for minimizing the live state of the GPU, thereby making it more efficient for context switches and speculative execution.

Prior efforts that dealt with divergences using hardware mechanisms performed their optimizations as and when they encountered divergences. Thus, they performed warp regrouping (control-flow) or sub-warp execution (memory divergence) when they encountered the divergence. Shadow Engine is unique in that it proposes to identify the divergence early and make scheduling decisions that lead up to the divergent code, which then can be optimized more efficiently.

4.8 Chapter Summary

In this chapter, we present a software-assisted hardware mechanism, Shadow Engines, that can mitigate the effects of control-flow and memory divergences in GPU architectures, and hence, improve performance. Shadow Engines identifies the points of divergences in an application as early as possible during execution, and appropriately takes action through thread regrouping and locality-aware warp scheduling to minimize the impact of control-flow and memory divergences, respectively. The required compiler support and hardware modifications to the GPU architecture are presented to facilitate the mitigation of both types of divergences. Simulation results with 32 GPU workloads show that our proposed integrated mechanism is able to improve performance, on an average, by 25.9% compared to the baseline design, and is much more effective than addressing any of the two types of divergences in isolation.

Chapter 5

Conclusions and Future Work

5.1 Summary of Dissertation Contributions

As the demand for increased processing power within a given power budget exacerbates, all major technology driven industries have started employing GPUs in various different forms. With the explosion of data, they have become an integral component from wearable computing to run machine learning workloads to warehouse-scale computing for scientific and high performance applications. Therefore, it has become imperative to push the performance and energy-efficiency envelopes of GPUs.

Furthermore, as data movement overheads severely limit scalability, it has become imperative to have designs that are *aware of data movement*. This dissertation is an effort in making throughput processors, specifically GPUs, more aware of data movement. It proposes compute scheduling techniques to improve performance and energy efficiency in GPU architectures with processing in memory capabilities. It also develops computation offloading mechanisms to reduce on-chip data movement and improve performance and energy efficiency. Finally, it also proposes techniques to mitigate the effects of control-flow and memory divergence in GPUs via efficient warp formation and scheduling. The

research conducted in this dissertation has **three major contributions**.

First, it proposed two new code scheduling techniques that enable effective use of processing-in-memory (PIM) mechanisms in PIM-Assisted GPU architectures, where a conventional GPU is augmented with 3D memory stacks that house simpler GPUs in their logic layers. First, a kernel offloading mechanism that accurately decides what code portions to offload to the GPUs in the 3D memory stack, using a new regression-based kernel affinity prediction model. Second, a concurrent kernel management mechanism that uses the affinity prediction model, a new kernel execution time prediction model, and kernel dependency information to decide which kernels to schedule concurrently on both the main GPU cores and the GPU cores in memory stacks.

Second, it proposed two complementary computation offloading techniques for minimizing on-chip data movement in GPU architectures, and hence, improve performance and energy efficiency. The first technique enables computational offloading to the LLCs, while the second technique complements the first technique by adding offloading capability to any node in the interconnect. It identifies several small and basic instruction chains in GPU applications that can be offloaded to any one of the locations.

Lastly, it proposed a novel software-assisted hardware technique, called Shadow Engines, to reduce the control-flow and memory divergence that arises at runtime. It shows that by identifying the divergence points early on in the execution, it is possible to take corrective actions and mitigate the effects of the divergences when they arise. Shadow Engines can, at runtime, dynamically form new warps, keeps track of memory divergence and prioritizes warps for scheduling in order to improve the formation of new warps as well as the ordering of the memory accesses to improve the consumption of the data from divergent memory operations.

5.2 Future Research Directions

One of the biggest challenges in computing systems is the need for high performance, high energy-efficiency, flexible programming and supporting a rich feature set such as virtualization, multiple application execution, etc. While throughput processors are being utilized ubiquitously, from wearables to warehouse-scale computing, they still suffer from a limited set of features and limited application-specific scalability. Due to the growing demand for high performance and energy efficient throughput processors, possible exciting future directions include 1) extending the use cases of shadow engines for various optimizations such as instruction and data prefetchers, branch predictors, etc.; 2) designing architectures with high degree of heterogeneity to improve performance and energy-efficiency in high-performance computing systems; 3) designing throughput architectures that can efficiently handle the execution of the rapidly changing landscape of machine learning applications; and 4) designing throughput processors that improve the security guarantees while minimizing performance impact.

5.2.1 Using Early Execution to Resolve Different Challenges

In Chapter 4, to efficiently mitigate the effects of control-flow and memory divergences, we proposed Shadow Engines. Shadow Engines make use of early execution, which helps identify and process the divergence points early. The mechanism of hoisting the code and executing a shadow version of an instruction has potentially far more use-cases than simply resolving control-flow and memory divergence in GPUs. It can potentially be used for providing accurate addresses to prefetchers, train branch predictors, help speculative execution in executing appropriate instruction streams, allow efficient control of nested parallelism in GPUs, etc. In this context, to extend Shadow Engines, we need to answer some key questions: 1) What modifications and information does shadow engines need for the above mentioned optimizations in GPUs? 2) How shadow engines can be ported

efficiently to other general purpose architectures such as CPUs to perform similar optimizations?

5.2.2 Heterogeneous Computing

With the demands for improving performance and energy-efficiency of domain-specific application on throughput processors increasing day by day, it has become inevitable to add heterogeneity throughout the computing stack starting from multiple different types of memory (NVM, DRAM, SRAM, etc.) and different processing elements (video decoder, tensor cores, ray tracing units, etc.). As per proof of concept in Chapter 2, that studies a GPU architecture with high-throughput, low-bandwidth main GPU and low-throughput, high bandwidth GPU-PIM, exploring the design space for further heterogeneity in GPUs is a promising research direction. In this context, some key questions that need to be answered are 1) how much and what type of heterogeneity is required? 2) how to map computations to which unit and at what granularity? and 3) how to maximize the performance and energy-efficiency of these architectures using near data computing techniques?

5.2.3 Accelerating Machine Learning Kernels using Near-Data Techniques on Throughput Processors

With the advent of high-throughput architectures, machine learning (ML) applications have become feasible and are being deployed for various different purposes such as natural language processing (NLP), data analytics [230], video analytics [231], IoT devices [232], gaming [233] and smart grid power management [234]. Therefore, there is a growing trend to add support for such domain-specific applications to improve performance and energy-efficiency. Since, machine learning applications involve high amounts of data movement, there are many near data computing optimizations that needs to be

investigated. In this context, some key questions that need to be answered are 1) what are the key computation bottlenecks in machine learning applications? 2) what computations (pre-processing, etc.) can be done closer to memory? and 3) how to take advantage of sparsity machine learning models in GPUs?

5.2.4 Improving Security in Throughput Processors

Recently, throughput processors are penetrating the entire computing system environment, from handheld devices, cloud gaming to executing scientific applications. Specifically, there has been a growing push for GPUs in public data centers for use by machine learning applications, financial simulation models and gaming. Since, resources in a data center need to be virtualized for cost-effectiveness and energy-efficiency, it has become imperative to support the execution of multiple applications in GPUs. Therefore, with multiple applications residing on the same GPU, the security capabilities of GPUs needs to be investigated. In this context, some key questions that need to be answered are 1) how vulnerable (side-channel attack [235], speculation attacks [236], etc.) are today's state-of-the-art GPUs when co-locating malicious applications with other applications, 2) how can techniques (if present) from CPU domain be ported to GPU to efficiently mitigate the attacks, and 3) how prior GPU optimization techniques affect the security of a GPU system and what are the ways to easily mitigate them?

Bibliography

[1] PTX, N. C. (2008) "Parallel thread execution," *NVIDIA Corp., Jun.*

[2] Obama, B. (2015) "Executive order-creating a national strategic computing initiative," *The White House, US*, **29**.

[3] Schulte, M. J., M. Ignatowski, G. H. Loh, B. M. Beckmann, W. C. Brantley, S. Gurumurthi, N. Jayasena, I. Paul, S. K. Reinhardt, and G. Rodgers (2015) "Achieving Exascale Capabilities through Heterogeneous Computing," *IEEE Micro*, **35**(4), pp. 26–36.

[4] Park, S. I., S. P. Ponce, J. Huang, Y. Cao, and F. Quek (2008) "Low-cost, high-speed computer vision using NVIDIA's CUDA architecture," in *Applied Imagery Pattern Recognition Workshop, 2008. AIPR'08. 37th IEEE*, IEEE, pp. 1–7.

[5] Eklund, A., P. Dufort, D. Forsberg, and S. M. LaConte (2013) "Medical Image Processing on the GPU-Past, Present and Future," *Medical Image Analysis*.

[6] Pratx, G. and L. Xing (2011) "GPU computing in medical physics: A review," *Medical physics*, **38**, p. 2685.

[7] Stone, S. S., J. P. Haldar, S. C. Tsao, W. mei W. Hwu, B. P. Sutton, and Z.-P. Liang (2008) "Accelerating advanced MRI reconstructions on GPUs," *J. Parallel Distrib. Comput.*, **68**(10), pp. 1307–1318.

[8] Foster, R. (2012) "How to harness big data for improving public health," *Government Health IT*.

[9] Gutierrez, E., S. Romero, M. Trenas, and E. Zapata (2007) "Simulation of quantum gates on a novel GPU architecture," in *Proceedings of the 7th Conference on 7th WSEAS International Conference on Systems Theory and Scientific Computation - Volume 7*, ISTASC'07, World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, pp. 121–126.

[10] Pagès, G. and B. Wilbertz (2012) "GPGPUs in computational finance: Massive parallel computing for American style options," *Concurrency and Computation: Practice and Experience*, **24**(8), pp. 837–848.

[11] Schmerken, I. (2009) "Wall street accelerates options analysis with GPU technology," *http://wallstreetandtech. com/technology-risk-management/showArticle. jhtml*.

[12] NVIDIA (2011) "JP Morgan Speeds Risk Calculations with NVIDIA GPUs," *Newsroom*.

[13] ———, "Researchers Deploy GPUs to Build World's Largest Artificial Neural Network," .

[14] ——— (2018), "Programming Guide," .

[15] TOP500 (2015), "Top500 Supercomputer Sites - June 2015," .

[16] Green500 (2015), "The Green500 List - June 2015," .

[17] Seneviratne, S., Y. Hu, T. Nguyen, G. Lan, S. Khalifa, K. Thilakarathna, M. Hassan, and A. Seneviratne (2017) "A survey of wearable devices and challenges," *IEEE Communications Surveys & Tutorials*, **19**(4), pp. 2573–2620.

[18] Cheng, K.-T. and Y.-C. Wang (2011) "Using mobile GPU for general-purpose computing–a case study of face recognition on smartphones," in *Proceedings of 2011 International Symposium on VLSI Design, Automation and Test*, IEEE, pp. 1–4.

[19] Holton, J. and T. Fratangelo (2012) "Raspberry Pi Architecture," *Raspberry Pi Foundation, London, UK*.

[20] NVIDIA (2011), "NVIDIA DRIVE - Scalable AI Platform for Autonomous Driving," .

[21] Ahmed, E., I. Yaqoob, I. A. T. Hashem, I. Khan, A. I. A. Ahmed, M. Imran, and A. V. Vasilakos (2017) "The role of big data analytics in Internet of Things," *Computer Networks*, **129**, pp. 459–471.

[22] Barbera, M. V., S. Kosta, A. Mei, and J. Stefa (2013) "To Offload or Not To Offload? The Bandwidth and Energy Costs of Mobile Cloud Computing," in *2013 Proceedings IEEE INFOCOM*, IEEE, pp. 1285–1293.

[23] Mazumdar, A., T. Moreau, S. Kim, M. Cowan, A. Alaghi, L. Ceze, M. Oskin, and V. Sathe (2017) "Exploring computation-communication tradeoffs in camera systems," in *2017 IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, pp. 177–186.

[24] Group, K. O. W. (2008) "The opencl specification," *A. Munshi, Ed.*

[25] NVIDIA (2017), "NVIDIA TESLA V100 GPU Architecture: The World's Most Advanced Data Center GPU," .

[26] AMD (2017), "Radeon next-generation Vega architecture ," .

[27] NVIDIA (2012), "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," .

[28] Kaeli, D. R., P. Mistry, D. Schaa, and D. P. Zhang (2015) *Heterogeneous computing with OpenCL 2.0*, Morgan Kaufmann.

[29] Kayiran, O., N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das (2014) "Managing GPU Concurrency in Heterogeneous Architectures," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE, pp. 114–126.

[30] Zhang, D., N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski (2014) "TOP-PIM: Throughput-oriented Programmable Processing in Memory," in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, ACM, pp. 85–98.

[31] Arunkumar, A., E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, C.-J. Wu, and D. Nellans (2017) "MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, ACM, New York, NY, USA, pp. 320–332.

[32] Nickolls, J. and W. J. Dally (2010) "The GPU computing era," *Micro, IEEE*, **30**(2), pp. 56–69.

[33] Russell, R. M. (1978) "The CRAY-1 Computer System," *Commun. ACM*, **21**(1), pp. 63–72.

[34] Colwell, R. P., R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman (1987) "A VLIW Architecture for a Trace Scheduling Compiler," in *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ASPLOS II, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 180–192.

[35] AMD (2012), "Graphics Cores Next (GCN) Architecture," .

[36] Rixner, S. (2002) *Stream Processor Architecture*, Kluwer Academic Publishers, Norwell, MA, USA.

[37] Kapasi, U. J., W. J. Dally, S. Rixner, J. D. Owens, and B. Khailany (2002) "The Imagine Stream Processor," in *Proceedings. IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 282–288.

[38] Dally, W. J. (2015) "Challenges for Future Computing Systems," *HiPEAC Keynote*.

[39] Keckler, S., W. Dally, B. Khailany, M. Garland, and D. Glasco (2011) "GPUs and the Future of Parallel Computing," *IEEE Micro*, pp. 7–17.

[40] Pattnaik, A., X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das (2016) "Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation Techniques*, ACM, pp. 31–44.

[41] Pattnaik, A., X. Tang, O. Kayiran, A. Jog, A. Mishra, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das (2019) "Opportunistic Computing in GPU Architectures," in *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, ACM, New York, NY, USA, pp. 210–223.

[42] Mutlu, O. (2013) "Memory scaling: A systems architecture perspective," in *2013 5th IEEE International Memory Workshop*, pp. 21–25.

[43] Moscibroda, T. and O. Mutlu (2007) "Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems," in *Proceedings of 16th USENIX Security*

Symposium on USENIX Security Symposium, SS'07, USENIX Association, Berkeley, CA, USA, pp. 18:1–18:18.

[44] Mutlu, O. and T. Moscibroda (2007) "Stall-time Fair Memory Access Scheduling for Chip Multiprocessors," in *Proceedings of the 40th Annual IEEE/ACM international Symposium on Microarchitecture*, IEEE Computer Society, pp. 146–160.

[45] ——— (2008) "Parallelism-aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, IEEE Computer Society, Washington, DC, USA, pp. 63–74.

[46] Kim, Y., D. Han, O. Mutlu, and M. Harchol-Balter (2010) "ATLAS: A Scalable and High-performance Scheduling Algorithm for Multiple Memory Controllers," in *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, IEEE, pp. 1–12.

[47] Kim, Y., M. Papamichael, O. Mutlu, and M. Harchol-Balter (2010) "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, pp. 65–76.

[48] Muralidhara, S. P., L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda (2011) "Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ACM, pp. 374–385.

[49] Subramanian, L., V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu (2013) "MISE: Providing performance predictability and improving fairness in shared main memory systems," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, pp. 639–650.

[50] Das, R., R. Ausavarungnirun, O. Mutlu, A. Kumar, and M. Azimi (2013) "Application-to-Core Mapping Policies to Reduce Memory System Interference in Multi-Core Systems," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, pp. 107–118.

[51] Kim, H., D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar (2014) "Bounding Memory Interference Delay in COTS-based Multi-Core Systems,"

in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, IEEE, pp. 145–154.

[52] Subramanian, L., V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu (2015) "The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory," in *Proceedings of the 48th International Symposium on Microarchitecture*, ACM, pp. 62–75.

[53] Subramanian, L., D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu (2014) "The Blacklisting Memory Scheduler: Achieving High Performance and Fairness at Low Cost," in *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, IEEE, pp. 8–15.

[54] Jog, A., O. Kayiran, A. Pattnaik, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das (2016) "Exploiting Core Criticality for Enhanced GPU Performance," in *ACM SIGMETRICS Performance Evaluation Review*, ACM, pp. 351–363.

[55] Ausavarungnirun, R., S. Ghose, O. Kayiran, G. H. Loh, C. R. Das, M. T. Kandemir, and O. Mutlu (2015) "Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, IEEE, pp. 25–38.

[56] Vijaykumar, N., G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, C. Das, M. Kandemir, T. C. Mowry, and O. Mutlu (2015) "A Case for Core-Assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, ACM, New York, NY, USA, pp. 41–53.

[57] Wang, H., C. Isci, L. Subramanian, J. Choi, D. Qian, and O. Mutlu (2015) "A-DRM: Architecture-aware Distributed Resource Management of Virtualized Clusters," in *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '15, ACM, New York, NY, USA, pp. 93–106.

[58] Lee, C. J., V. Narasiman, O. Mutlu, and Y. N. Patt (2009) "Improving Memory Bank-level Parallelism in the Presence of Prefetching," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, pp. 327–336.

[59] Kayiran, O., A. Jog, M. T. Kandemir, and C. R. Das (2013) "Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, IEEE Press, pp. 157–166.

[60] Gokhale, M., B. Holmes, and K. Iobst (1995) "Processing in Memory: the Terasys Massively Parallel PIM Array," *IEEE Computer*.

[61] Draper, J., J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, and G. Daglikoca (2002) "The Architecture of the DIVA Processing-in-memory Chip," in *Proceedings of the 16th international conference on Supercomputing*, ACM, pp. 14–25.

[62] Ahn, J., S. Hong, S. Yoo, O. Mutlu, and K. Choi (2015) "A Scalable Processing-in-memory Accelerator for Parallel Graph Processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, ACM, New York, NY, USA, pp. 105–117.

[63] Ahn, J., S. Yoo, O. Mutlu, and K. Choi (2015) "PIM-enabled Instructions: A Low-overhead, Locality-aware Processing-In-Memory Architecture," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, IEEE, pp. 336–348.

[64] Balasubramonian, R., J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson (2014) "Near-Data Processing: Insights from a MICRO-46 Workshop," in *IEEE Micro*, IEEE, pp. 36–42.

[65] Stone, H. S. (1970) "A Logic-in-Memory Computer," *IEEE Transactions on Computers*, pp. 73–78.

[66] Kogge, P. M. (1994) "EXECUBE-A New Architecture for Scaleable MPPs," in *1994 International Conference on Parallel Processing Vol. 1*, vol. 1, IEEE, pp. 77–84.

[67] Lipovski, G. J. and C. Yu (1999) "The Dynamic Associative Access Memory Chip and Its Application to SIMD Processing and Full-Text Database Retrieval," in *Records of the 1999 IEEE International Workshop on Memory Technology, Design and Testing*, IEEE, pp. 24–31.

[68] Patterson, D., T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick (1997) "A Case for Intelligent RAM," *IEEE micro*, pp. 34–44.

[69] Hall, M., P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park (1999) "Mapping Irregular Applications to DIVA, a PIM-based Data-Intensive Architecture," in *SC'99: Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, IEEE, pp. 57–57.

[70] Kang, Y., W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas (1999) "FlexRAM: Toward an Advanced Intelligent Memory System," in *Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors (Cat. No. 99CB37040)*, IEEE, pp. 192–201.

[71] Loh, G. H. (2008) "3D-Stacked Memory Architectures for Multi-core Processors," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, IEEE Computer Society, Washington, DC, USA, pp. 453–464.

[72] Woo, D. H., N. H. Seong, D. Lewis, and H.-H. Lee (2010) "An Optimized 3D-Stacked Memory Architecture by Exploiting Excessive, High-Density TSV Bandwidth," in *The Sixteenth International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 1–12.

[73] Pawlowski, J. T. (2011) "Hybrid Memory Cube," *Hotchips*.

[74] Lee, D., S. Ghose, G. Pekhimenko, S. Khan, and O. Mutlu (2016) "Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost," *ACM Transactions on Architecture and Code Optimization (TACO)*, pp. 63:1–63:29.

[75] JEDEC (2013) *JESD235 High Bandwidth Memory (HBM) DRAM*.

[76] Hsieh, K., E. Ebrahimi, G. Kim, N. Chatterjee, M. OConnor, N. Vijaykumar, O. Mutlu, and S. W. Keckler (2016) "Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, IEEE Press, Piscataway, NJ, USA, pp. 204–216.

[77] NVIDIA (2011), "Fermi: NVIDIA's Next Generation CUDA Compute Architecture," .

[78] Ausavarungnirun, R., K. K.-W. Chang, L. Subramanian, G. H. Loh, and O. Mutlu (2012) "Staged Memory Scheduling: Achieving High Prformance

and Scalability in Heterogeneous Systems," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, IEEE Computer Society, Washington, DC, USA, pp. 416–427.

[79] Usui, H., L. Subramanian, K. K.-W. Chang, and O. Mutlu (2016) "DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators," *ACM Transactions on Architecture and Code Optimization (TACO)*, **12**(4), pp. 65:1–65:28.

[80] Zhao, J. and Y. Xie (2012) "Optimizing Bandwidth and Power of Graphics Memory with Hybrid Memory Technologies and Adaptive Data Migration," in *2012 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 81–87.

[81] Liu, J., B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu (2013) "An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, ACM, New York, NY, USA, pp. 60–71.

[82] Liu, J., B. Jaiyen, R. Veras, and O. Mutlu (2012) "RAIDR: Retention-Aware Intelligent DRAM Refresh," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, IEEE Computer Society, Washington, DC, USA, pp. 1–12.

[83] Chang, K., D. Lee, Z. Chishti, A. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu (2014) "Improving DRAM Performance by Parallelizing Refreshes with Accesses," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, pp. 356–367.

[84] Qureshi, M., D. H. Kim, S. Khan, P. Nair, and O. Mutlu (2015) "AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, IEEE, pp. 427–437.

[85] Jog, A., O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das (2013) "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance," in *Proceedings of the Eighteenth International Conference on Architectural Support*

for Programming Languages and Operating Systems, ASPLOS '13, ACM, New York, NY, USA, pp. 395–406.

[86] Eckert, Y., N. Jayasena, and G. H. Loh (2014) "Thermal Feasibility of Die-Stacked Processing in Memory," in *WoNDP*, pp. 1–5.

[87] Danalis, A., G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter (2010) "The Scalable Heterogeneous Computing (SHOC) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, ACM, New York, NY, USA, pp. 63–74.

[88] Leng, J., T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi (2013) "GPUWattch: Enabling Energy Optimizations in GPGPUs," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, ACM, New York, NY, USA, pp. 487–498.

[89] NVIDIA (2015), "NVIDIA's Next Generation CUDA Compute Architecture: Maxwell GM20x," .

[90] Goswami, N., R. Shankar, M. Joshi, and T. Li (2010) "Exploring GPGPU Workloads: Characterization Methodology, Analysis and Microarchitecture Evaluation Implications," in *IEEE International Symposium on Workload Characterization (IISWC'10)*, IEEE, pp. 1–10.

[91] Burtscher, M., R. Nasre, and K. Pingali (2012) "A Quantitative Study of Irregular Programs on GPUs," in *2012 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 141–151.

[92] NVIDIA (2011), "CUDA C/C++ SDK Code Samples," .

[93] Grauer-Gray, S., L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos (2012) "Auto-tuning a High-level Language targeted to GPU Codes," in *2012 Innovative Parallel Computing (InPar)*, IEEE, pp. 1–10.

[94] Che, S., M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron (2009) "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, IEEE, pp. 44–54.

[95] Buluç, A., J. R. Gilbert, and C. Budak (2010) "Solving Path Problems on the GPU," *Parallel Computing*.

[96] He, B., W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang (2008) "Mars: A MapReduce Framework on graphics processors," in *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 260–269.

[97] Karlin, I., A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. Devito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. H. Still (2013) "Exploring Traditional and Emerging Parallel Programming Models Using a Proxy Application," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pp. 919–932.

[98] Hosmer, D. W. and S. Lemeshow (2000) *Applied Logistic Regression*, John Wiley & Sons.

[99] Fan, R.-E., K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin (2008) "LIBLINEAR: A Library for Large Linear Classification," *Journal of machine learning research*, pp. 1871–1874.

[100] Montgomery, D. C. and E. Peck (1992) *Introduction to Linear Regression Analysis*, John Wiley & Sons.

[101] Dubach, C., J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, and O. Temam (2007) "Fast Compiler Optimisation Evaluation Using Code-Feature Based Performance Prediction," in *Proceedings of the 4th International Conference on Computing Frontiers*, CF '07, ACM, New York, NY, USA, pp. 131–142.

[102] Garey, M. R., D. S. Johnson, and L. Stockmeyer (1974) "Some Simplified NP-complete Problems," in *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, STOC '74, ACM, New York, NY, USA, pp. 47–63.

[103] Bakhoda, A., G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt (2009) "Analyzing CUDA workloads using a detailed GPU simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, IEEE, pp. 163–174.

[104] Jog, A., O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. Keckler, M. T. Kandemir, and C. R. Das (2015) "Anatomy of GPU Memory System for Multi-Application Execution," in *Proceedings of the 2015 International*

Symposium on Memory Systems, MEMSYS '15, ACM, New York, NY, USA, pp. 223–234.

[105] Rogers, T. G., M. O'Connor, and T. M. Aamodt (2013) "Divergence-aware Warp Scheduling," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, ACM, New York, NY, USA, pp. 99–110.

[106] ——— (2012) "Cache-Conscious Wavefront Scheduling," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, IEEE Computer Society, Washington, DC, USA, pp. 72–83.

[107] GPGPU-Sim v3.2.1 ( ) "GTX 480 Configuration," .

[108] ——— ( ) "Address mapping," .

[109] Jevdjic, D., G. Loh, C. Kaynak, and B. Falsafi (2014) "Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 25–37.

[110] Reddaway, S. F. (1973) "DAP - a Distributed Array Processor," in *Proceedings of the 1st Annual Symposium on Computer Architecture*, ACM, pp. 61–65.

[111] Sayre, G. E. (1976) "STARAN: An Associative Approach to Multiprocessor Architecture," in *Computer Architecture, Workshop of the Gesellschaft fur Informatik*, pp. 199–221.

[112] Smitley, D. and K. Iobst (1991) "Bit-Serial SIMD on the CM-2 and the Cray 2," *Journal of Parallel and Distributed Computing*, pp. 135 – 145.

[113] Carter, J., W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama (1999) "Impulse: Building a Smarter Memory Controller," in *Proceedings Fifth International Symposium on High-Performance Computer Architecture*, IEEE, pp. 70–79.

[114] Rahimi, A., A. Ghofrani, M. A. Lastras-Montano, K.-T. Cheng, L. Benini, and R. K. Gupta (2014) "Energy-Efficient GPGPU Architectures via Collaborative Compilation and Memristive Memory-Based Computing," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, IEEE, pp. 1–6.

[115] Seshadri, V., Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry (2013) "RowClone: Fast and Energy-efficient in-DRAM Bulk Data Copy and Initialization," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ACM, pp. 185–197.

[116] Seshadri, V., K. Hsieh, A. Boroum, D. Lee, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry (2015) "Fast Bulk Bitwise AND and OR in DRAM," *IEEE Computer Architecture Letters*, **14**(2), pp. 127–131.

[117] Seshadri, V., T. Mullins, A. Boroumand, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry (2015) "Gather-Scatter DRAM: In-DRAM Address Translation to Improve the Spatial Locality of Non-unit Strided Accesses," in *Proceedings of the 48th International Symposium on Microarchitecture*, ACM, pp. 267–280.

[118] Scrbak, M., M. Islam, K. M. Kavi, M. Ignatowski, and N. Jayasena (2015) "Processing-in-Memory: Exploring the Design Space," in *International Conference on Architecture of Computing Systems*, Springer, pp. 43–54.

[119] Guo, Q., N. Alachiotis, B. Akin, F. Sadi, G. Xu, T. M. Low, L. Pileggi, J. C. Hoe, and F. Franchetti (2014) "3D-Stacked Memory-Side Acceleration: Accelerator and System Design," in *In the Workshop on Near-Data Processing (WoNDP)(Held in conjunction with MICRO-47.)*, pp. 1–6.

[120] Pugsley, S. H., J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li (2014) "NDC: Analyzing the Impact of 3D-Stacked Memory+Logic Devices on MapReduce Workloads," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, pp. 190–200.

[121] Boroumand, A., S. Ghose, B. Lucia, K. Hsieh, K. Malladi, H. Zheng, and O. Mutlu (2016) "LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory," *IEEE Computer Architecture Letters*, pp. 46–50.

[122] Loh, G. H., N. Jayasena, M. H. Oskin, M. Nutter, D. Roberts, M. Meswani, D. P. Zhang, and M. Ignatowski (2013) "A Processing-in-Memory Taxonomy and a Case for Studying Fixed-function PIM," in *Workshop on Near-Data Processing (WoNDP)*, pp. 1–4.

[123] Hashemi, M., Khubaib, E. Ebrahimi, O. Mutlu, and Y. N. Patt (2016) "Accelerating Dependent Cache Misses with an Enhanced Memory Controller," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, IEEE, pp. 444–455.

[124] Zhang, H., S. Zhao, A. Pattnaik, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das (2019) "Distilling the Essence of Raw Video to Reduce Memory Usage and Energy at Edge Devices," in *Proceedings of the 52th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-52 '19, p. .

[125] Farmahini-Farahani, A., J. Ahn, K. Compton, and N. Kim (2014) "DRAMA: An Architecture for Accelerated Processing near Memory," *IEEE Computer Architecture Letters*, pp. 26–29.

[126] Ma, K., X. Li, W. Chen, C. Zhang, and X. Wang (2012) "GreenGPU: A Holistic Approach to Energy Efficiency in GPU-CPU Heterogeneous Architectures," in *2012 41st International Conference on Parallel Processing*, IEEE, pp. 48–57.

[127] Zhang, L., B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang (2010) "Accurate Online Power Estimation and Automatic Battery Behavior based Power Model Generation for Smartphones," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, ACM, pp. 105–114.

[128] Li, T. and L. K. John (2003) "Run-time Modeling and Estimation of Operating System Power Consumption," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, ACM, pp. 160–171.

[129] Nagasaka, H., N. Maruyama, A. Nukada, T. Endo, and S. Matsuoka (2010) "Statistical Power Modeling of GPU Kernels using Performance Counters," in *International conference on green computing*, IEEE, pp. 115–122.

[130] Bailey, P. E., D. K. Lowenthal, V. Ravi, B. Rountree, M. Schulz, and B. R. De Supinski (2014) "Adaptive Configuration Selection for Power-constrained Heterogeneous Systems," in *2014 43rd International Conference on Parallel Processing*, IEEE, pp. 371–380.

[131] Wu, G., J. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou (2015) "GPGPU Performance and Power Estimation using Machine Learning," in *2015 IEEE*

21st International Symposium on High Performance Computer Architecture (HPCA), pp. 564–576.

[132] Ardalani, N., C. Lestourgeon, K. Sankaralingam, and X. Zhu (2015) "Cross-Architecture Performance Prediction (XAPP) Using CPU Code to Predict GPU Performance," in *Proceedings of the 48th International Symposium on Microarchitecture*, ACM, pp. 725–737.

[133] Panwar, L. S., A. M. Aji, J. Meng, P. Balaji, and W.-c. Feng (2013) "Online Performance Projection for Clusters with Heterogeneous GPUs," in *2013 International Conference on Parallel and Distributed Systems*, IEEE, pp. 283–290.

[134] Ipek, E., B. R. De Supinski, M. Schulz, and S. A. McKee (2005) "An Approach to Performance Prediction for Parallel Applications," in *European Conference on Parallel Processing*, Springer, pp. 196–205.

[135] Ïpek, E., S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz (2006) "Efficiently Exploring Architectural Design Spaces via Predictive Modeling," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, ACM, New York, NY, USA, pp. 195–206.

[136] Ipek, E., O. Mutlu, J. F. Martnez, and R. Caruana (2008) "Self Optimizing Memory Controllers: A Reinforcement Learning Approach," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, pp. 39–50.

[137] Xu, C. and F. C. Lau (1996) *Load Balancing in Parallel Computers: Theory and Practice*, Springer Science & Business Media.

[138] Cybenko, G. (1989) "Dynamic load balancing for distributed memory multiprocessors," *Journal of parallel and distributed computing*, pp. 279–301.

[139] Shirazi, B. A., K. M. Kavi, and A. R. Hurson (eds.) (1995) *Scheduling and Load Balancing in Parallel and Distributed Systems*, IEEE Computer Society Press, Los Alamitos, CA, USA.

[140] Gregg, C., M. Boyer, K. Hazelwood, and K. Skadron (2011) "Dynamic Heterogeneous Scheduling Decisions using Historical Runtime Data," in *Workshop on Applications for Multi-and Many-Core Processors (A4MMC)*, pp. 1–12.

[141] Chandra, R., S. Devine, B. Verghese, A. Gupta, and M. Rosenblum (1994) "Scheduling and Page Migration for Multiprocessor Compute Servers," in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 12–24.

[142] Sharma, A., H. Jiang, A. Pattnaik, J. Kotra, M. T. Kandemir, and C. R. Das (2019) "CASH: Compiler Assisted Hardware Design for Improving DRAM Energy Efficiency in CNN Inference," in *Proceedings of the 2019 International Symposium on Memory Systems (MEMSYS)*, ACM, p. .

[143] Augonnet, C., S. Thibault, R. Namyst, and P.-A. Wacrenier (2011) "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience*.

[144] Topcuoglu, H., S. Hariri, and M.-y. Wu (2002) "Performance-effective and Low-complexity Task Scheduling for Heterogeneous Computing," *IEEE transactions on parallel and distributed systems*, pp. 260–274.

[145] Joao, J. A., M. A. Suleman, O. Mutlu, and Y. N. Patt (2012) "Bottleneck Identification and Scheduling in Multithreaded Applications," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 223–234.

[146] ——— (2013) "Utility-based Acceleration of Multithreaded Applications on Asymmetric CMPs," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pp. 154–165.

[147] Suleman, M. A., O. Mutlu, M. K. Qureshi, and Y. N. Patt (2009) "Accelerating Critical Section Execution with Asymmetric Multi-core Architectures," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, ACM, New York, NY, USA, pp. 253–264.

[148] Suleman, M. A., O. Mutlu, J. A. Joao, Khubaib, and Y. N. Patt (2010) "Data Marshaling for Multi-core Architectures," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, ACM, New York, NY, USA, pp. 441–450.

[149] Aji, A. M., A. J. Pena, P. Balaji, and W.-c. Feng (2015) "Automatic Command Queue Scheduling for Task-Parallel Workloads in OpenCL," in *2015 IEEE International Conference on Cluster Computing*, IEEE, pp. 42–51.

[150] Chen, L., O. Villa, S. Krishnamoorthy, and G. R. Gao (2010) "Dynamic Load Balancing on Single-and Multi-GPU Systems," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, IEEE, pp. 1–12.

[151] Sharifi, A., E. Kultursay, M. Kandemir, and C. R. Das (2012) "Addressing end-to-end memory access latency in noc-based multicores," in *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, IEEE, pp. 294–304.

[152] Adhinarayanan, V., I. Paul, J. L. Greathouse, W. Huang, A. Pattnaik, and W. c. Feng (2016) "Measuring and modeling on-chip interconnect power on real hardware," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 1–11.

[153] Bakhoda, A., J. Kim, and T. M. Aamodt (2010) "Throughput-effective on-chip networks for manycore accelerators," in *Proceedings of the 2010 43rd Annual IEEE/ACM international symposium on Microarchitecture*, IEEE Computer Society, pp. 421–432.

[154] Kim, K. H., R. Boyapati, J. Huang, Y. Jin, K. H. Yum, and E. J. Kim (2017) "Packet Coalescing Exploiting Data Redundancy in GPGPU Architectures," in *Proceedings of the International Conference on Supercomputing*, ICS '17, ACM, New York, NY, USA, pp. 6:1–6:10.

[155] Dublish, S., V. Nagarajan, and N. Topham (2017) "Evaluating and mitigating bandwidth bottlenecks across the memory hierarchy in GPUs," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 239–248.

[156] ——— (2016) "Characterizing memory bottlenecks in GPGPU workloads," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 1–2.

[157] Zhan, J., O. Kayiran, G. H. Loh, C. R. Das, and Y. Xie (2016) "OSCAR: Orchestrating STT-RAM cache traffic for heterogeneous CPU-GPU architectures," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–13.

[158] Dally, W. J. and B. Towles (2001) "Route Packets, Not Wires: On-chip Inteconnection Networks," in *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, ACM, New York, NY, USA, pp. 684–689.

[159] Carr, S., K. S. McKinley, and C.-W. Tseng (1994) *Compiler optimizations for improving data locality*, ACM.

[160] Zhang, E. Z., Y. Jiang, Z. Guo, K. Tian, and X. Shen (2011) "On-the-fly Elimination of Dynamic Irregularities for GPU Computing," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, ACM, New York, NY, USA, pp. 369–380.

[161] Fung, W. W., I. Sham, G. Yuan, and T. M. Aamodt (2007) "Dynamic warp formation and scheduling for efficient GPU control flow," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, pp. 407–420.

[162] Tang, X., O. Kislal, M. Kandemir, and M. Karakoy (2017) "Data Movement Aware Computation Partitioning," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, ACM, New York, NY, USA, pp. 730–744.

[163] Sun, C., C.-H. O. Chen, G. Kurian, L. Wei, J. Miller, A. Agarwal, L.-S. Peh, and V. Stojanovic (2012) "DSENT-a tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling," in *Networks on Chip (NoCS), 2012 Sixth IEEE/ACM International Symposium on*, IEEE, pp. 201–210.

[164] Muralimanohar, N., R. Balasubramonian, and N. P. Jouppi (2009) "CACTI 6.0: A tool to model large caches," *HP Laboratories*, pp. 22–31.

[165] Fung, W. W. and T. M. Aamodt (2011) "Thread block compaction for efficient SIMT control flow," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, IEEE, pp. 25–36.

[166] Kayiran, O., A. Jog, A. Pattnaik, R. Ausavarungnirun, X. Tang, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das (2016) "μC-States: Fine-grained GPU datapath power management," in *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, IEEE, pp. 17–30.

[167] Tang, X., A. Pattnaik, H. Jiang, O. Kayiran, A. Jog, S. Pai, M. Ibrahim, M. T. Kandemir, and C. R. Das (2017) "Controlled kernel launch for dynamic parallelism in GPUs," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, pp. 649–660.

[168] Tang, X., A. Pattnaik, O. Kayiran, A. Jog, M. T. Kandemir, and C. Das (2018) "Quantifying Data Locality in Dynamic Parallelism in GPUs," in *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, ACM, p. 39.

[169] Meng, J., D. Tarjan, and K. Skadron (2010) "Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, ACM, New York, NY, USA, pp. 235–246.

[170] Anantpur, J. and R. Govindarajan (2017) "Taming warp divergence," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*, pp. 50–60.

[171] Nai, L., R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim (2017) "GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 457–468.

[172] Aji, A. M., A. J. Peña, P. Balaji, and W.-c. Feng (2016) "MultiCL: Enabling Automatic Scheduling for Task-Parallel Workloads in OpenCL," *Parallel Computing*.

[173] Thinakaran, P., J. R. Gunasekaran, B. Sharma, M. T. Kandemir, and C. R. Das (2017) "Phoenix: A constraint-aware scheduler for heterogeneous datacenters," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, IEEE, pp. 977–987.

[174] Rengasamy, P. V., H. Zhang, N. Nachiappan, S. Zhao, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das (2017) "Characterizing diverse handheld apps for customized hardware acceleration," in *2017 IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, pp. 187–196.

[175] Hadidi, R., L. Nai, H. Kim, and H. Kim (2017) "CAIRO: A Compiler-Assisted Technique for Enabling Instruction-Level Offloading of Processing-In-Memory," *ACM Trans. Archit. Code Optim.*, **14**(4), pp. 48:1–48:25.

[176] Jang, H., J. Kim, P. Gratz, K. H. Yum, and E. J. Kim (2015) "Bandwidth-efficient On-chip Interconnect Designs for GPGPUs," in *Proceedings of the 52nd Annual Design Automation Conference*, DAC '15, ACM, New York, NY, USA, pp. 9:1–9:6.

[177] Gottlieb, A., R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir (1982) "The NYU Ultracomputer&Mdash;Designing a MIMD, Shared-memory Parallel Machine (Extended Abstract)," in *Proceedings of the 9th Annual Symposium on Computer Architecture*, ISCA '82, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 27–42.

[178] Boroumand, A., S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, and O. Mutlu (2018) "Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, ACM, New York, NY, USA, pp. 316–331.

[179] Kim, H., H. Kim, S. Yalamanchili, and A. F. Rodrigues (2015) "Understanding Energy Aspects of Processing-near-Memory for HPC Workloads," in *Proceedings of the 2015 International Symposium on Memory Systems*, MEMSYS '15, ACM, New York, NY, USA, pp. 276–282.

[180] Tennenhouse, D. L., J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden (1997) "A survey of active network research," *Communications Magazine, IEEE*, **35**(1), pp. 80–86.

[181] Mishra, A. K., N. Vijaykrishnan, and C. R. Das (2011) "A case for heterogeneous on-chip interconnects for CMPs," in *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pp. 389–400.

[182] Ziabari, A. K., J. L. Abellán, Y. Ma, A. Joshi, and D. Kaeli (2015) "Asymmetric NoC Architectures for GPU Systems," in *Proceedings of the 9th International Symposium on Networks-on-Chip*, NOCS '15, ACM, New York, NY, USA, pp. 25:1–25:8.

[183] Michelogiannakis, G., N. Jiang, D. Becker, and W. J. Dally (2011) "Packet Chaining: Efficient Single-cycle Allocation for On-chip Networks," in *Proceedings of*

the 44th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-44, ACM, New York, NY, USA, pp. 83–94.

[184] Ramakrishna, M., V. K. Kodati, P. V. Gratz, and A. Sprintson (2016) "GCA:Global Congestion Awareness for Load Balance in Networks-on-Chip," *IEEE Transactions on Parallel and Distributed Systems*, **27**(7), pp. 2022–2035.

[185] Ramanujam, R. S. and B. Lin (2010) "Destination-based adaptive routing on 2D mesh networks," in *2010 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pp. 1–12.

[186] ATI (2010) "ATI Radeon GPGPUs," .

[187] Fatahalian, K., J. Sugerman, and P. Hanrahan (2004) "Understanding the efficiency of GPU algorithms for matrix-matrix multiplication," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ACM, pp. 133–137.

[188] Volkov, V. and J. W. Demmel (2008) "Benchmarking GPUs to tune dense linear algebra," in *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, IEEE, pp. 1–11.

[189] Sadrosadati, M., A. Mirhosseini, S. B. Ehsani, H. Sarbazi-Azad, M. Drumond, B. Falsafi, R. Ausavarungnirun, and O. Mutlu (2018) "LTRF: Enabling High-Capacity Register Files for GPUs via Hardware/Software Cooperative Register Prefetching," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, ACM, New York, NY, USA, pp. 489–502.

[190] Ausavarungnirun, R., V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. J. Rossbach, and O. Mutlu (2018) "MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, New York, NY, USA, pp. 503–518.

[191] Yoon, H., J. Lowe-Power, and G. S. Sohi (2018) "Filtering Translation Bandwidth with Virtual Caching," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, New York, NY, USA, pp. 113–127.

[192] Luebke, D., M. Harris, N. Govindaraju, A. Lefohn, M. Houston, J. Owens, M. Segal, M. Papakipos, and I. Buck (2006) "GPGPU: general-purpose computation on graphics hardware," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, ACM, p. 208.

[193] Giunta, G., R. Montella, G. Agrillo, and G. Coviello (2010) "A GPGPU transparent virtualization component for high performance computing clouds," in *European Conference on Parallel Processing*, Springer, pp. 379–391.

[194] Takada, N., T. Shimobaba, N. Masuda, and T. Ito (2009) "High-speed FDTD simulation algorithm for GPU with compute unified device architecture," in *2009 IEEE Antennas and Propagation Society International Symposium*, IEEE, pp. 1–4.

[195] O'Neil, M. A. and M. Burtscher (2014) "Microarchitectural performance characterization of irregular GPU kernels," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, pp. 130–139.

[196] Wang, B., W. Yu, X.-H. Sun, and X. Wang (2015) "Dacache: Memory divergence-aware gpu cache management," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ACM, pp. 89–98.

[197] Zhang, E. Z., Y. Jiang, Z. Guo, and X. Shen (2010) "Streamlining GPU applications on the fly: thread divergence elimination through runtime thread-data remapping," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ACM, pp. 115–126.

[198] Brunie, N., S. Collange, and G. Diamos (2012) "Simultaneous branch and warp interweaving for sustained GPU performance," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, pp. 49–60.

[199] AMD (2019), "RDNA Architecture," .

[200] ARM (2019) "New Valhall architecture for Arm Mali-G77 GPU brings big step-change in premium mobile," *https://community.arm.com/developer/tools-software/graphics/b/blog/posts/introducing-arm-mali-g77-with-new-valhall-architecture*.

[201] Muchnick, S. S. (1997) *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[202] Rengasamy, P. V., H. Zhang, S. Zhao, N. C. Nachiappan, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das (2018) "CritICs Critiquing Criticality in Mobile Apps," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, pp. 867–880.

[203] Bacon, D. F., S. L. Graham, and O. J. Sharp (1994) "Compiler transformations for high-performance computing," *ACM Computing Surveys (CSUR)*, **26**(4), pp. 345–420.

[204] NVIDIA (2011), "NVIDIA CUDA Toolkit," .

[205] Lee, M., S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu (2014) "Improving GPGPU resource utilization through alternative thread block scheduling," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, pp. 260–271.

[206] Chatterjee, N., M. OConnor, D. Lee, D. R. Johnson, S. W. Keckler, M. Rhu, and W. J. Dally (2017) "Architecting an energy-efficient dram system for gpus," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, pp. 73–84.

[207] Giles, M. (2008), "Jacobi iteration for a Laplace discretisation on a 3D structured grid," .

[208] Schatz, M. C., C. Trapnell, A. L. Delcher, and A. Varshney (2007) "High-throughput sequence alignment using Graphics Processing Units," *BMC bioinformatics*, **8**(1), p. 474.

[209] Michalakes, J. and M. Vachharajani (2008) "GPU acceleration of numerical weather prediction," *Parallel Processing Letters*, **18**(04), pp. 531–548.

[210] Che, S., B. M. Beckmann, S. K. Reinhardt, and K. Skadron (2013) "Pannotia: Understanding irregular GPGPU graph applications," in *2013 IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, pp. 185–195.

[211] Stratton, J. A., C. Rodrigues, I. J. Sung, N. Obeid, L. W. Chang, N. Anssari, G. D. Liu, and W. W. Hwu (2012) *Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing*, *Tech. Rep. IMPACT-12-01*, University of Illinois, at Urbana-Champaign.

[212] Vaidya, A. S., A. Shayesteh, D. H. Woo, R. Saharoy, and M. Azimi (2013) "SIMD Divergence Optimization Through Intra-warp Compaction," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pp. 368–379.

[213] Hong, S., S. K. Kim, T. Oguntebi, and K. Olukotun (2011) "Accelerating CUDA Graph Algorithms at Maximum Warp," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, ACM, New York, NY, USA, pp. 267–276.

[214] Narasiman, V., M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt (2011) "Improving GPU performance via large warps and two-level warp scheduling," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ACM, pp. 308–317.

[215] Han, T. D. and T. S. Abdelrahman (2011) "Reducing branch divergence in GPU programs," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, ACM, p. 3.

[216] Xiang, P., Y. Yang, and H. Zhou (2014) "Warp-level divergence in GPUs: Characterization, impact, and mitigation," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, pp. 284–295.

[217] Anantpur, J. and R. Govindarajan (2014) "Taming Control Divergence in GPUs through Control Flow Linearization," in *Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, pp. 133–153.

[218] Sartori, J. and R. Kumar (2012) "Branch and data herding: Reducing control and memory divergence for error-tolerant GPU applications," *IEEE Transactions on Multimedia*, **15**(2), pp. 279–290.

[219] Lee, S.-Y. and C.-J. Wu (2014) "CAWS: Criticality-aware Warp Scheduling for GPGPU Workloads," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, ACM, New York, NY, USA, pp. 175–186.

[220] Lee, S.-Y., A. Arunkumar, and C.-J. Wu (2015) "CAWA: Coordinated Warp Scheduling and Cache Prioritization for Critical Warp Acceleration of GPGPU

Workloads," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ACM, New York, NY, USA, pp. 515–527.

[221] Lee, M., G. Kim, J. Kim, W. Seo, Y. Cho, and S. Ryu (2016) "iPAWS: Instruction-issue pattern-based adaptive warp scheduling for GPGPUs," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, pp. 370–381.

[222] Rogers, T. G., D. R. Johnson, M. O'Connor, and S. W. Keckler (2015) "A Variable Warp Size Architecture," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ACM, New York, NY, USA, pp. 489–501.

[223] Kim, K., S. Lee, M. K. Yoon, G. Koo, W. W. Ro, and M. Annavaram (2016) "Warped-preexecution: A GPU pre-execution approach for improving latency hiding," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, pp. 163–175.

[224] Liu, S., C. Eisenbeis, and J.-L. Gaudiot (2011) "Value Prediction and Speculative Execution on GPU," *International Journal of Parallel Programming*, **39**(5), pp. 533–552.

[225] Diamos, G. and S. Yalamanchili (2010) "Speculative execution on multi-GPU systems," in *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pp. 1–12.

[226] Menon, J., M. De Kruijf, and K. Sankaralingam (2012) "iGPU: Exception Support and Speculative Execution on GPUs," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, IEEE Computer Society, Washington, DC, USA, pp. 72–83.

[227] Mutlu, O., J. Stark, C. Wilkerson, and Y. N. Patt (2003) "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, IEEE, pp. 129–140.

[228] Tran, K.-A., A. Jimborean, T. E. Carlson, K. Koukos, M. Själander, and S. Kaxiras (2018) "SWOOP: Software-hardware Co-design for Non-speculative, Execute-ahead, In-order Cores," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, ACM, New York, NY, USA, pp. 328–343.

[229] Sakalis, C., S. Kaxiras, A. Ros, A. Jimborean, and M. Själander (2019) "Efficient Invisible Speculative Execution Through Selective Delay and Value Prediction," in *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, ACM, New York, NY, USA, pp. 723–735.

[230] Kelleher, J. D., B. Mac Namee, and A. D'arcy (2015) *Fundamentals of machine learning for predictive data analytics: algorithms, worked examples, and case studies*, MIT Press.

[231] Ananthanarayanan, G., P. Bahl, P. Bodík, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha (2017) "Real-time video analytics: The killer app for edge computing," *Computer*, **50**(10), pp. 58–67.

[232] Li, H., K. Ota, and M. Dong (2018) "Learning IoT in edge: deep learning for the internet of things with edge computing," *IEEE Network*, **32**(1), pp. 96–101.

[233] NVIDIA (2018), "NVIDIA DLSS - Deep Learning Super Sampling," .

[234] Rudin, C., D. Waltz, R. N. Anderson, A. Boulanger, A. Salleb-Aouissi, M. Chow, H. Dutta, P. N. Gross, B. Huang, S. Ierome, D. F. Isaac, A. Kressner, R. J. Passonneau, A. Radeva, and L. Wu (2012) "Machine learning for the New York City power grid," *IEEE transactions on pattern analysis and machine intelligence*, **34**(2), pp. 328–345.

[235] Naghibijouybari, H., A. Neupane, Z. Qian, and N. Abu-Ghazaleh (2018) "Rendered insecure: GPU side channel attacks are practical," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ACM, pp. 2139–2153.

[236] Islam, S., A. Moghimi, I. Bruhns, M. Krebbel, B. Gulmezoglu, T. Eisenbarth, and B. Sunar (2019) "SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks," *arXiv preprint arXiv:1903.00446*.

Vita

Ashutosh Pattnaik

Ashutosh Pattnaik was born in Bhubaneswar, India, in 1992. Ashutosh holds a bachelor's degree in Electronics and Instrumentation Engineering from the National Institute of Technology, Rourkela. Ashutosh joined Penn State as a Ph.D. student in 2013, and worked on GPU architectures under the supervisions of Dr. Chita Das, Dr. Mahmut Taylan Kandemir and Dr. Anand Sivasubramaniam. He has served as a technical reviewer for many journals including IEEE TPDS, IEEE TCC, Microprocessors and Microsystems and ETRI Journal. He served as a teaching assistant for Logic Design, and Computer Architecture courses. His research has been published in top-tier computer architecture conferences such as MICRO, HPCA, ISCA, and PACT. Ashutosh worked as an intern at AMD Research in 2015 and 2016.