Managing the Memory Hierarchy in GPUs

Saumay Dublish

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2018

Abstract

Pervasive use of GPUs across multiple disciplines is a result of continuous adaptation of the GPU architectures to address the needs of upcoming application domains. One such vital improvement is the introduction of the on-chip cache hierarchy, used primarily to filter the high bandwidth demand to the off-chip memory. However, in contrast to traditional CPUs, the cache hierarchy in GPUs is presented with significantly different challenges such as cache thrashing and bandwidth bottlenecks, arising due to small caches and high levels of memory traffic. These challenges lead to severe congestion across the memory hierarchy, resulting in high memory access latencies. In memory-intensive applications, such high memory access latencies often get exposed and can no longer be hidden through multithreading, and therefore adversely impact system performance.

In this thesis, we address the inefficiencies across the memory hierarchy in GPUs that lead to such high levels of congestion. We identify three major factors contributing to poor memory system performance: first, disproportionate and insufficient bandwidth resources in the cache hierarchy; second, poor cache management policies; and third, high levels of multithreading. In order to revitalize the memory hierarchy by addressing the above limitations, we propose a three-pronged approach. First, we characterize the bandwidth bottlenecks present across the memory hierarchy in GPUs and identify the architectural parameters that are most critical in alleviating congestion. Subsequently, we explore the architectural design space to mitigate the bandwidth bottlenecks in a cost-effective manner. Second, we identify significant inter-core reuse in GPUs, presenting an opportunity to reuse data among the L1s. We exploit this reuse by connecting the L1 caches with a lightweight ring network to facilitate inter-core communication of shared data. We show that this technique reduces traffic to the L2 cache, freeing up the bandwidth for other accesses. Third, we present *Poise*, a machine learning approach to mitigate cache thrashing and bandwidth bottlenecks by altering the levels of multithreading. *Poise* comprises a supervised learning model that is trained offline on a set of profiled kernels to make good warp scheduling decisions. Subsequently, a hardware inference engine is used to predict good warp scheduling decisions at runtime using the model learned during training. In summary, we address the problem of bandwidth bottlenecks across the memory hierarchy in GPUs by exploring how to best *scale*, *supplement* and *utilize* the existing bandwidth resources. These techniques provide an effective and comprehensive methodology to mitigate the bandwidth bottlenecks in the GPU memory hierarchy.

Lay Summary

Our day-to-day life revolves around the marvels of computational progress. Ranging from small-scale devices such as smart phones and fitness trackers to large-scale services such as search engines and cloud computing — all are rooted in the tremendous progress made by the computer industry. Over the last decade, a lot of this progress is credited to the success of *Graphics Processing Units* or GPUs. Recent advancements have led to significant improvements in the graphics performance, which is evidenced by high resolution games and realistic animation films. In addition, modern GPUs have also greatly influenced general-purpose areas such as artificial intelligence and healthcare.

Modern GPUs pose several challenges to computer architects. These challenges can be explained through a *restaurant* analogy. In older times, restaurant kitchens lacked modern storage equipments to preserve perishable products. Therefore, chefs had to repeatedly visit the distant grocery stores to buy raw materials for their recipes. With the advent of in-house storage facilities such as refrigerators, number of trips to the grocery stores could be reduced by storing or *caching* perishable groceries more proximately. A problem arises when large number of chefs pose conflicting storage needs. In such a scenario, shared storage proves to be insufficient in terms of capacity as well as the available parallelism in accessing the shared storage. This results in large queues and congestion at the refrigerators (and even at the grocery stores). For a given amount of storage and floorspace, the congestion problem can be addressed primarily in three ways. Firstly, by allowing multiple chefs to access the storage at the same time; for instance, by adding independent doors to each shelf in the refrigerator. Secondly, by encouraging chefs to cooperatively share common products to avoid redundant trips to the refrigerator by multiple chefs for the same items. Thirdly, by limiting the number of chefs so that their storage demand matches the available kitchen resources, while also ensuring constant food supply to the customers. Similar to the restaurant scenario, where multiple chefs process raw food by following a recipe, GPUs have multiple *processors* that crunch raw data by following a set of instructions. GPUs retrieve this data from storage elements such as proximate caches and distant memories, and then pour it into the computational machinery inside the GPU to complete a task. Similar to food stores, shared memory resources in GPUs also suffer from severe congestion. In this thesis, we address the high levels of congestion in GPUs and propose three solutions, analogous to the solutions proposed for the restaurant problem. In effect, we investigate ways to efficiently manage the shared memory resources in GPUs.

Acknowledgements

I would like to express my deep gratitude to my advisor, Prof. Nigel Topham, for his guidance, advice and kindness throughout my graduate studies. Nigel gave me the freedom and encouragement to explore new problems, which was vital in enabling me to mature as a researcher. He always had the time and utmost patience whenever I needed advice and direction in my research. I am extremely grateful for his mentorship.

I would like to thank my co-advisor, Dr. Vijay Nagarajan, for his invaluable counsel. His confidence in my abilities and constant encouragement during the highs and lows of my graduate studies kept me motivated, and inspired me to pursue meaningful research. The door to his office was always open for discussions and brainstorming, which proved extremely useful in shaping my work. I am very grateful to him for all the guidance and support.

I would like to thank Dr. Boris Grot for being part of the annual review committee and providing valuable advice and suggestions. I also thank Prof. Michael O'Boyle and Dr. Robert Mullins for being on my viva panel and making it a memorable experience. I thank several members of ICSA who provided valuable feedback, suggestions and a space to improve my ideas. I would like to specially thank Arpit Joshi, Cheng-Chieh Huang, José Cano Reyes, Marco Elver, Priyank Faldu and Rakesh Kumar for their valuable help and support, both technical and otherwise. I also thank several anonymous reviewers from the Computer Architecture community who provided valuable feedback and suggestions, which was crucial in improving my work. Furthermore, I am grateful for the School of Informatics PhD Scholarship and Saranu International Research Scholarship in enabling my PhD study.

I would like to thank all my friends who offered me their joyous company during my time in Edinburgh. Their friendship helped me cruise through the years of my graduate studies, and made each and every moment spent here memorable.

Finally, I would like to express my profound gratitude to my parents, Geeta Dublish and Rajeev Dublish, for providing me with the values, guidance and unwavering support throughout my life. It is difficult to imagine reaching this juncture without their enormous love and sacrifices. I also thank my brother, Tushar Dublish, for his incredible support and assistance throughout my graduate studies.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Saumay Dublish*)

Dedicated to my parents

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the past decade, the journey of GPUs in transforming the computing landscape has been phenomenal. Initially used to render graphics for computer games, GPUs are now at the centre of major advancements in areas ranging from artificial intelligence to enterprise computing. As a result, GPUs have proven to be of substantive significance in the world of general-purpose computing, emerging as a formidable alternative to traditional single-threaded processors.

Wide contrast in the performance of CPUs and GPUs derives from a fundamental difference in the design philosophies of these architectures. CPUs are heavily optimized for single-threaded performance, dedicating a substantial portion of the die area to storage units, such as caches, for faster data access. In contrast, GPUs are heavily multithreaded and dedicate most of the die area to compute units, making them more tolerant to memory latencies of individual threads. In such multithreaded architectures, however, memory bandwidth poses a much bigger challenge as high levels of multi-threading present a high demand on the off-chip memory bandwidth. This has led to the introduction of deeper memory hierarchies in modern GPUs, comprising multilevel caches, to filter the bandwidth demand to off-chip memory. However, compared to CPUs, the cache hierarchy in GPUs encounter significantly different demands and bottlenecks due to considerably higher volumes of memory traffic. As a result, the adoption of conventional cache hierarchy in GPUs has triggered significant research efforts to address these new bottlenecks and to ensure a well performing memory system, which is crucial for the future success of GPUs.

1.1 The Problem

The caches in GPUs are much smaller in comparison to CPU caches, in line with the throughput-oriented nature of GPUs. For instance, Intel's Haswell processor [55] features over 8 MB of on-chip memory that is shared by a handful of threads. In contrast, NVIDIA's Kepler GPU [116] features only around 2 MB of on-chip memory that is shared by tens of thousands of threads. Consequently, small caches and large number of threads in GPUs can lead to high cache miss rates and cache thrashing [66, 67, 132]. Due to poor cache performance, the cache hierarchy does not adequately filter the bandwidth demand to off-chip memory. In addition, the cache hierarchy exposes its own bandwidth limitations in sustaining such high levels of memory traffic [139]. As a result, the bandwidth implications of the entire memory hierarchy, including caches, pose a severe challenge to GPU performance and warrants investigation [76]. In the remainder of this section, we briefly present the principal factors that lead to bandwidth bottlenecks across the memory hierarchy in GPUs and discuss their performance implications. Later, in subsequent chapters, we delve into more details about these challenges.

1.1.1 Inadequate Bandwidth Resources

A typical memory hierarchy comprises small and fast first-level caches closest to the processors, and progressively larger and slower caches at the lower levels of the memory hierarchy. The aggregate bandwidth provided by each memory level also tapers as we go lower in the memory hierarchy, and is depicted by the *bandwidth pyramid* in Figure 1.1. This is because each level filters the bandwidth demand to the lower level, and consequently the lower levels require only a fraction of bandwidth of the higher levels. Therefore, appropriate allocation of bandwidth resources to each level of the memory hierarchy, *i.e.*, shaping the bandwidth pyramid, is critical for a well-performing memory system.

In GPUs, the off-chip bandwidth continues to scale with upcoming memory technologies such as High Bandwidth Memories (HBM), providing high bandwidth to inject memory responses into the cache hierarchy. In addition, the number of GPU cores continue to grow, increasing the bandwidth demand on the cache hierarchy. However, the conventional cache hierarchy adopted by GPUs is under-provisioned in bandwidth resources (such as on-chip interconnect, peripheral buffers, *etc.*) to handle such high volumes of memory traffic. As a result, the bandwidth hierarchy is becoming increasingly imbalanced, leading to growing congestion across the memory hierarchy — specifically

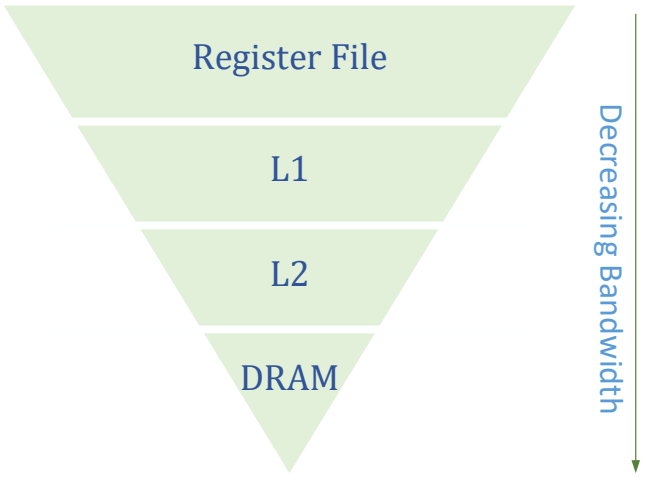


Figure 1.1: Typical bandwidth hierarchy in GPUs

the caches. Therefore, a key challenge is to investigate the distribution of bandwidth demand and the resultant bottlenecks across the memory hierarchy in GPUs, and evaluate the design space to provision the cache hierarchy with appropriate bandwidth resources, thereby mitigating the bandwidth bottlenecks.

1.1.2 Poor Cache Management

Typically, GPUs exhibit high cache miss rates, indicating that the cache performance is far from perfect. This is partly because current GPU cache management techniques are unable to utilize these caches effectively. One such inefficiency in current GPU cache management policies is the repeated access to the shared L2 cache from different L1 caches for the same data, arising due to inter-core data reuse. Such a policy is a common occurrence in CPUs and is generally benign for performance when L1 miss rates are low. In such cases, it does not excessively deplete the memory bandwidth. In GPUs, however, such a policy is corrosive to the overall performance due to high L1 miss rates. As a result, for memory-intensive applications, where performance is constrained by memory bandwidth, such a policy aggravates the bandwidth issue by repeated memory requests for data already cached elsewhere at the same level in the memory hierarchy. Evidently, the policies that are suitable for CPUs may not be suitable for GPUs. Therefore, it is critical to address the inefficiencies of the existing cache management policies in GPUs to ensure effective usage of scarce bandwidth resources. Notably, the inefficient usage of caches in GPUs is even more alarming considering that the die area they occupy might otherwise be used to implement additional GPU

cores, which arguably hold much more importance to throughput-oriented computing systems. For this to be an effective trade-off, the efficiency of the cache hierarchy must be maximized.

1.1.3 High Thread-level Parallelism

GPUs employ a Single-Instruction Multiple-Threads (SIMT) execution model, which requires a large number of threads to execute a set of similar instructions. Due to the abundance of parallel work available in such systems, set of threads that encounter a long latency memory operation are replaced by another set of threads that are ready to resume execution, while the pending memory operations are completed. Therefore, GPUs rely on a high degree of thread-level parallelism (via multithreading) to hide the long latency of memory operations. Often, in compute-intensive applications where long latency memory operations are infrequent, increasing the number of threads available for multithreading continues to improve the latency hiding ability of GPUs. However, in memory-intensive applications where memory instructions account for a significant fraction of the total instructions, increasing the number of threads often leads to a considerable increase in demand for memory resources such as caches and memory bandwidth. Consequently, the high demand for memory resources overwhelms the memory hierarchy and leads to poor memory system performance, giving rise to problems such as cache thrashing and bandwidth bottlenecks. As a result, these side-effects adversely affect system performance, thereby diminishing the benefits of parallelism.

The problems of cache thrashing and bandwidth bottlenecks can be mitigated by reducing the level of multithreading in the GPUs. However, the optimal level of multithreading may vary significantly across different applications and architectures, and therefore naively restricting multithreading can adversely affect throughput. Due to this tension between *thread-level parallelism* and *memory system performance*, and its direct impact on cache performance and bandwidth demand, balancing these two properties poses a significant challenge.

1.1.4 Implications of Bandwidth Bottlenecks

Due to the bandwidth bottlenecks arising out of the above challenges, there can be severe congestion across the memory hierarchy in GPUs. Such high levels of congestion lead to increased average memory latencies, and this has three major implications. Firstly, in

memory-intensive applications, due to insufficient computation to mask high memory latencies, such latencies appear in the critical path of system performance. Secondly, high latencies of outstanding miss requests lead to prolonged contention for cache resources such as Miss Status Holding Registers (MSHRs) and replaceable cache lines. This effect increases memory latencies even further, as succeeding requests get serialized and have to wait for outstanding misses to complete and relinquish the resources. And thirdly, back pressure from a congested lower level further throttles the cache pipeline and prevents it from operating at peak throughput, exacerbating the bandwidth limitation in the cache hierarchy. The combination of the above factors forces the cores to stall, leading to performance degradation.

1.1.5 Summary

In summary, we identify that the bandwidth bottleneck across the memory hierarchy is a significant problem in modern GPUs; this is the key issue that we address in this thesis. There are three principal factors that lead to bandwidth bottlenecks: firstly, the growing bandwidth imbalance in the memory hierarchy; secondly, inefficient cache management; and thirdly, high levels of thread-level parallelism. In memory-intensive applications, these factors lead to high congestion in the memory hierarchy, thereby adversely impacting performance.

1.2 Contributions

In this thesis, we propose the following three-pronged approach to address the bandwidth bottlenecks arising out of the issues discussed in the previous section. Firstly, we characterize the memory hierarchy with respect to the bandwidth bottlenecks and propose cost-effective scaling of bandwidth resources. Secondly, we propose a cooperative caching mechanism for L1 caches to improve the caching efficiency in GPUs with respect to the bandwidth demand on the lower levels. And thirdly, we propose a machine learning technique to adaptively balance thread-level parallelism and memory system performance. We discuss these strategies briefly in the remainder of this section, followed by detailed evaluation and discussion in the subsequent chapters.

1.2.1 Cost-effective Scaling of Bandwidth Resources

In this proposal (detailed in Chapter 3), we address the challenge of rising imbalance in the bandwidth hierarchy in GPUs, which is discussed in Section 1.1.1. We characterize the bandwidth bottlenecks present across the memory hierarchy in GPUs and quantify the stalls throughout the memory hierarchy. We use this characterization to identify the architectural parameters that are most critical in alleviating congestion. We explore the architectural design space to mitigate the bandwidth bottlenecks and show that the performance improvement achieved by mitigating the bandwidth bottleneck in the cache hierarchy can exceed the speedup obtained by a memory system with a baseline cache hierarchy and High Bandwidth Memory (HBM) DRAM. This signifies the importance of resolving bandwidth bottlenecks in the cache hierarchy, in comparison to simply increasing off-chip memory bandwidth.

We also show that addressing the bandwidth bottleneck in isolation, at specific levels, can be sub-optimal and can even be counter-productive. Therefore, we show that it is imperative to resolve the bandwidth bottlenecks synergistically across different levels of the memory hierarchy. With the insights developed in this proposal, we perform a cost-benefit analysis and identify cost-effective configurations of the memory hierarchy that effectively mitigate the bandwidth bottlenecks. We conclude this study by presenting a cost-effective configuration that comprises an asymmetric crossbar alongside scaled peripheral resources such as buffers and MSHRs. We show that our final configuration achieves a performance improvement of 29% on average, with a minimal area overhead of 1.5%, compared to the baseline modern GPU.

1.2.2 Cooperative Caching for L1 Caches

In this proposal (detailed in Chapter 4), we aim to address an inefficiency in the traditional cache management techniques, which is discussed in Section 1.1.2. We identify significant data reuse across different GPU cores, presenting an opportunity to reuse data among the L1 caches. By sharing data among L1s, we aim to reduce the pressure on the shared L2 bandwidth, thereby reducing the memory access latencies that lie in the critical path. We show how data reuse can be exploited via an L1 Cooperative Caching Network (CCN), thereby supplementing the existing bandwidth resources and reducing demand on the shared L2 bandwidth. In the proposed architecture, we connect the L1 caches with a lightweight ring network to facilitate inter-core communication of shared data. We show that this technique reduces traffic to the L2 cache by an average

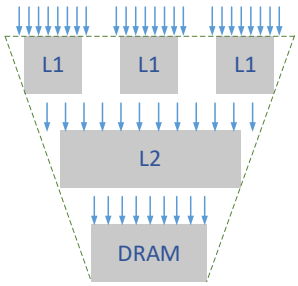
of 29%, freeing up the shared L2 bandwidth for other accesses. We also show that CCN reduces the average memory latency by 24%, thereby reducing core stall cycles by 26% on average. This translates into an overall performance improvement of 14.7% on average (and up to 49%) for applications that exhibit reuse across L1 caches. In doing so, CCN incurs a nominal area and energy overhead of 1.3% and 2.5% respectively. Notably, the performance improvement with our proposed CCN compares favourably to the performance improvement achieved by simply doubling the number of L2 banks by up to 34%.

1.2.3 Managing Thread-level Parallelism

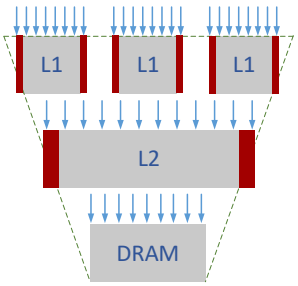
In this proposal (detailed in Chapter 5), we address the adverse effect of high thread-level parallelism on memory system performance, which is discussed in Section 1.1.3. We present *Poise*, a novel approach that alters the warp scheduling mechanism in the GPU to balance thread-level parallelism and memory system performance. *Poise* achieves this by adding a set of two knobs to the warp scheduler. The first knob determines the number of warps that can pollute the cache in order to improve memory system performance, while the second knob determines the overall thread-level parallelism in order to maximize the multithreading available in the system. *Poise* determines these warp scheduling decisions using the following two major components: a machine learning framework and a hardware inference engine. The machine learning framework comprises a supervised learning model that is trained offline on a large set of profiled kernels to make good warp scheduling decisions. This is achieved by learning a mapping from a set of application and architectural features, to the warp scheduling decisions that led to the best performance for a kernel. At runtime, the hardware inference engine collects these features periodically using hardware performance counters, and uses the mapping that was learned during training to dynamically predict good warp scheduling decisions. As a result, *Poise* achieves a performance improvement of up to 2.94$\times$ and a harmonic mean speedup of 46.6%, over the baseline greedy-then-oldest warp scheduler. It also outperforms the prior state-of-the-art warp scheduler by an average of 15.1%, while incurring minimal hardware overheads in contrast to prior techniques.

1.2.4 Summary

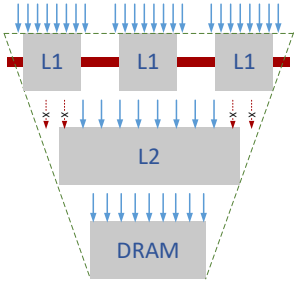
In this thesis, we investigate the bandwidth implications of the different levels of the memory hierarchy and demonstrate how the current policies breach the latency tolerance



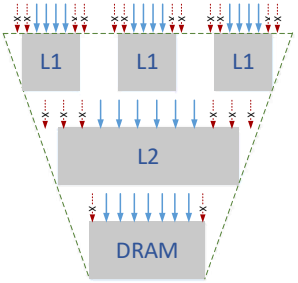
(a) Baseline bandwidth hierarchy when high bandwidth demand causes a bottleneck.



(b) *Scaling* the existing bandwidth resources in the cache hierarchy.



(c) *Supplementing* the existing bandwidth resources in the cache hierarchy.



(d) *Utilizing* the existing bandwidth resources by reducing the bandwidth demand.

Figure 1.2: Proposed three-pronged approach to address the bandwidth bottlenecks across the memory hierarchy in GPUs. Bandwidth demand is represented by *blue arrows*; reduction in bandwidth demand is represented by crossed *red arrows*; and scaled or additional bandwidth resources are represented by *red blocks*.

property of GPUs by causing congestion across the memory hierarchy. The baseline bandwidth hierarchy is represented in Figure 1.2a where high demand on the memory hierarchy leads to a bandwidth bottleneck. Note that we omit register files from the discussion as they are often underutilized for general-purpose applications [1, 121] and are seldom a bandwidth bottleneck. We propose a three-pronged approach to address the problem. Firstly, we investigate how to *best scale the existing bandwidth resources* in the GPU through characterization and design space exploration (Figure 1.2b). Secondly, we investigate how to *best supplement the existing bandwidth resources* through a cooperative caching network for L1 caches (Figure 1.2c). And thirdly, we investigate how to *best utilize the existing bandwidth resources* by balancing thread-level parallelism and memory system performance (Figure 1.2d).

1.3 Published Work

Some of the contents of this thesis have appeared in the following publications:

- S. Dublish, *Student Research Poster: Slack-Aware Shared Bandwidth Management in GPUs*, ACM SRC, The 25th International Conference on Parallel Architectures and Compilation Techniques (PACT), Haifa, Israel, September 11-15, 2016.

  — Appears in *Chapter 6 (Section 6.3)*.

- S. Dublish, V. Nagarajan and N. Topham, *Characterizing Memory Bottlenecks in GPGPU Workloads*, IEEE International Symposium on Workload Characterization (IISWC), Providence, Rhoda Island, USA, September 25-27, 2016.

  — Appears in *Chapter 3*.

- S. Dublish, V. Nagarajan and N. Topham, *Cooperative Caching for GPUs*, ACM Transactions on Architecture and Code Optimization (TACO), 13(4), 39, December 2016.

  — Appears in *Chapter 4*.

- S. Dublish, V. Nagarajan and N. Topham, *Evaluating and Mitigating Bandwidth Bottlenecks Across the Memory Hierarchy in GPUs*, IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Santa Rosa, USA, April 23-25, 2017.

  — Appears in *Chapter 3*.

1.4 Organization

The remainder of this thesis is organized as follows: Chapter 2 provides a background about GPU computing. Chapter 3 presents a characterization of bandwidth bottlenecks in the memory hierarchy of GPUs and evaluates the design space for cost-effective scaling of the bandwidth resources. Chapter 4 evaluates the reuse patterns for general-purpose applications and presents CCN, a Cooperative Caching Network for L1 caches in GPUs. Chapter 5 presents *Poise*, a warp scheduling policy to balance multithreading and memory system performance, consequently lowering the demand for shared

bandwidth resources in the memory hierarchy. Chapter 6 concludes the thesis by summarizing the findings and contributions of this work as well as exploring the scope for future work.

Chapter 2

GPU Computing

Consumer graphics hardware started gaining attention as early as the 1980s — at the time used primarily in arcade and console gaming devices. In the initial years, industry efforts were focused towards enabling 2D graphics through a variety of video cards and graphic adaptors [29, 48, 156]. Later, during the 1990s, further progress in the graphics industry ushered in an era of 3D graphics and revolutionized computer graphics. Throughout that decade, graphics hardware companies such as ATI (now acquired by AMD), 3Dfx and NVIDIA dominated most of the consumer graphics industry. With the turn of the century, NVIDIA introduced GeForce 256 [42] — the first commercially available Graphics Processing Unit or GPU. In 2006, after a series of architectural revamps and modifications, NVIDIA introduced the G80 series GPUs. It featured the Unified Shader architecture — an array of unified and similar compute hardware units in a GPU, in contrast to traditional graphics pipeline units with specialized functions such as rasterization and pixel shading [104] — adding momentum to the rise of general-purpose computing on GPUs, often known as GPGPU. The success of GPUs was fuelled further by the introduction of CUDA (maintained by NVIDIA) and OpenCL (maintained by the Khronos Group) — the parallel computing platform and programming models that bolstered the general-purpose ecosystem around GPUs. CUDA and OpenCL enabled GPUs to be used pervasively by the high performance computing community across different domains by allowing users to conveniently express the parallelism in their applications using these programming platforms. As a result, the current generation of GPUs have not only pushed the gaming industry forward by leaps and bounds, but have also emerged as a cornerstone of high performance computing for a variety of general-purpose application domains.

In this chapter, we delve into the finer details of modern GPU computing and provide

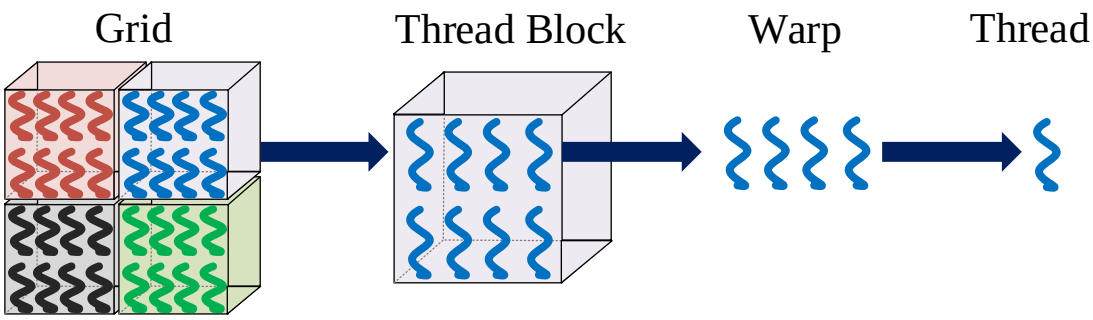


Figure 2.1: CUDA program structure

the necessary background for the remainder of this thesis. Note that we restrict our discussion to NVIDIA GPUs as these are arguably the most prevalent discrete GPUs in the industry. However, as GPUs from other vendors feature similar architectural features, memory system organization and programming models, we expect our observations to be applicable to other GPU architectures as well. In the remainder of this chapter, we first present the CUDA programming model in Section 2.1, followed by an overview of a modern NVIDIA GPU architecture in Section 2.2.

2.1 Programming Model

In a typical CPU-GPU system with a discrete GPU, the CPU is referred to as the *host* and executes the serial or modestly data parallel sections of the application, whereas the GPU is referred to as the *device* and executes the highly data parallel sections of the applications. CUDA — formerly an acronym for Compute Unified Device Architecture — is the standard parallel computing platform and programming model developed by NVIDIA for its GPUs [32]. Figure 2.1 depicts a typical program structure in CUDA. A CUDA program consists of data parallel structures called *kernels* that comprise tens of thousands of threads. The host CPU launches the kernel on the device GPU as a multidimensional *grid* of threads. Within a kernel, the large number of threads are organized into structured blocks of computation known as *thread blocks*. A device-level hardware scheduler in the GPU schedules the thread blocks to the different GPU cores for execution. The policy to schedule thread blocks on GPU cores can have a considerable impact on performance and has been evaluated in prior work [75, 96, 154, 155, 102].

The maximum number of thread blocks that can be co-scheduled on each GPU core at a given time is governed by the occupancy constraints specified by the vendor on

various system resources such as registers and threads. Therefore, the number of thread blocks available for multithreading is restricted by the demand for constrained resources posed by each thread block. Nevertheless, such a policy accords dynamism to GPUs in determining the number of thread blocks that can be scheduled concurrently on each GPU core, instead of restricting it to a constant number. As a result, it grants users the flexibility to examine the trade-off between a smaller number of thread blocks per GPU core, where each thread block is allotted higher resources, and a higher number of thread blocks per GPU core, where each thread block is allotted fewer resources. Expert programmers try to optimize their programs in order to maximize resource usage without significantly sacrificing the degree of multithreading available on each GPU core present in form of thread blocks [118, 83]. In addition, several mechanisms have been proposed to relieve the programmer from the burden of computing the best trade-off between the number of thread blocks and the resources allotted to each thread block [13, 134, 135, 152, 105, 14, 164, 159].

The different thread blocks in a kernel are executed independently and can be scheduled in any order. While threads within a thread block can be synchronized using a barrier synchronization primitive in CUDA called `__syncthreads()`, threads across different threads blocks cannot be directly synchronized using similar programming primitives. However, different thread blocks can communicate and collaborate implicitly through the global memory. In addition, synchronization across thread blocks can generally be enforced through the CPU by terminating and relaunching the kernel at the desired synchronization points. However, such coarse CPU-driven thread block synchronization occurs at a significant cost of additional communication between the host and the device [147]. Several recent proposals have discussed methodologies to extend or improve the existing synchronization primitives in GPUs through schemes pertaining to GPU barrier synchronization [41, 160, 147, 114, 163, 168] and scope synchronization [122, 142, 6], among others.

Each thread block consists of several smaller group of threads called *warps* — the smallest granularity for scheduling threads within a GPU core. The threads within a warp execute in a Single-Instruction Multiple-Threads (SIMT) fashion where the hardware functional units execute the same instruction across multiple threads of a warp in lockstep before moving to the next instruction. Instructions executed by the hardware are called SIMD instructions, *i.e.,* Single-Instruction Multiple-Data. SIMD instructions operate on a vector of data, instead of scalar data items. The length of the vector is known as the SIMD width.

In the case of control flow instructions, such as conditional branches, threads within a warp may take different execution routes after resolving their branch condition. This phenomenon is referred to as *warp divergence*. In such a scenario, hardware executes the divergent sets of threads sequentially until they converge and continue lockstep execution. High levels of control flow divergence often sacrifices the benefits of parallelism, and this has been addressed extensively in prior work [45, 113, 133, 39]. In most CUDA-enabled GPU architectures, warp divergence also prevents the divergent group of threads from communicating and sharing data due to their sequential execution. The recent Volta architecture, however, supports independent thread scheduling by maintaining per-thread scheduling resources (such as the program counter), instead of per-warp scheduling resources. This gives finer thread scheduling control and allows divergent threads to execute instructions in an interleaved fashion, enabling fine-grained communication and synchronization between threads within a divergent warp [108]. Volta still follows the SIMT execution model as the same instruction is executed by all active threads in a warp at any point in time, albeit with more flexible interleaving between divergent threads.

2.2 GPU Architecture

A typical GPU consists of several functional units organized as a set of highly multi-threaded and pipelined cores that are referred to as *Streaming Multiprocessors* or *SMs*.[1] Figure 2.2 depicts a system-level diagram of a typical GPU. Each SM comprises a unified shader pipeline consisting of fetch, decode, issue, execution and write-back stages. The fetch stage retrieves the warp instructions from the instruction buffers, while the decode stage resolves the operands and their associated dependencies. The dependency between operands is recorded through a *scoreboarding* mechanism. The scoreboard tracks the read-after-write (RAW) and write-after-write (WAW) dependency hazards within and across warps by reserving registers for the decoded destinations that require updating. If a subsequent instruction reads or writes a reserved register, a dependency is flagged, which is then resolved at the write-back stage on completion of the pending memory operation. Figure 2.3 illustrates these stages diagrammatically in greater detail. More architectural details can be found in [141].

[1] In this thesis, we use the terms “core” and “SM” interchangeably, when referring to GPUs.

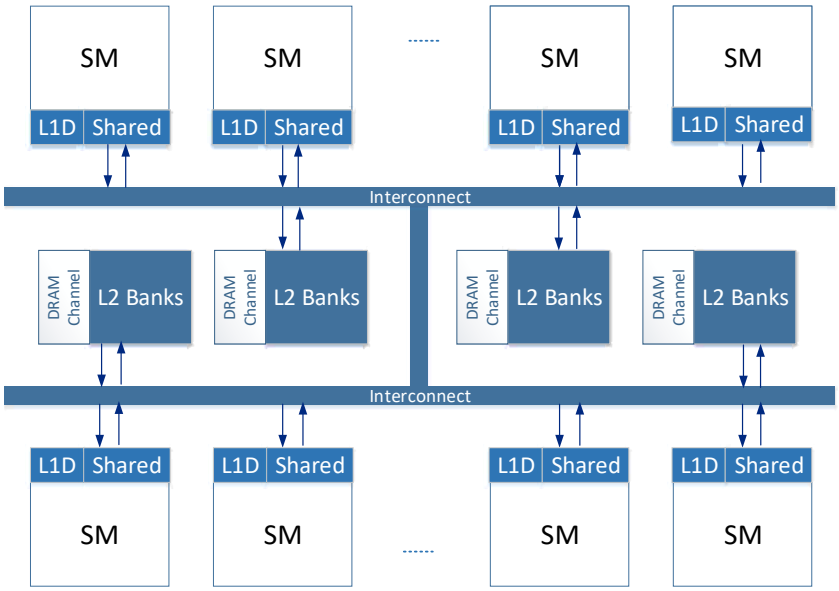


Figure 2.2: System-level architecture of a typical GPU

2.2.1 Warp Scheduling

Each SM consists of multiple hardware warp schedulers (not shown in the figures). After the decode stage, a warp scheduler selects a warp from a pool of *active warps* and issues them to the functional units, *i.e.,* floating-point unit (FPU) or load-store unit (LSU). A warp is termed active only when there exists a warp instruction that has all the operands required for execution and the scoreboard detects no dependency hazard with any outstanding memory operation. Once an active warp instruction is issued, it is scheduled and executed in lockstep on the appropriate functional units, depending on the instruction type. These functional units comprise multiple lanes of compute (floating-point) and memory (load-store) pipelines. Furthermore, if the scoreboard detects a dependency hazard with a pending memory operation, the corresponding warps are descheduled and forced to stall. Such warps are referred to as *inactive warps*. The inactive warps do not participate in multithreading until the pending memory operations are completed. In such a scenario, the warp scheduler selects a warp from the pool of active warps to replace the stalled inactive warp and allows continued execution, thereby overlapping the latency of pending memory operations. When the pending memory operations are completed, the missing operands are made available, thereby triggering the write-back stage for pending memory instructions. As a result, the formerly reserved registers are released and dependency hazards at the scoreboard are resolved. However, in the event when no active warps are available, the functional units are forced to stall

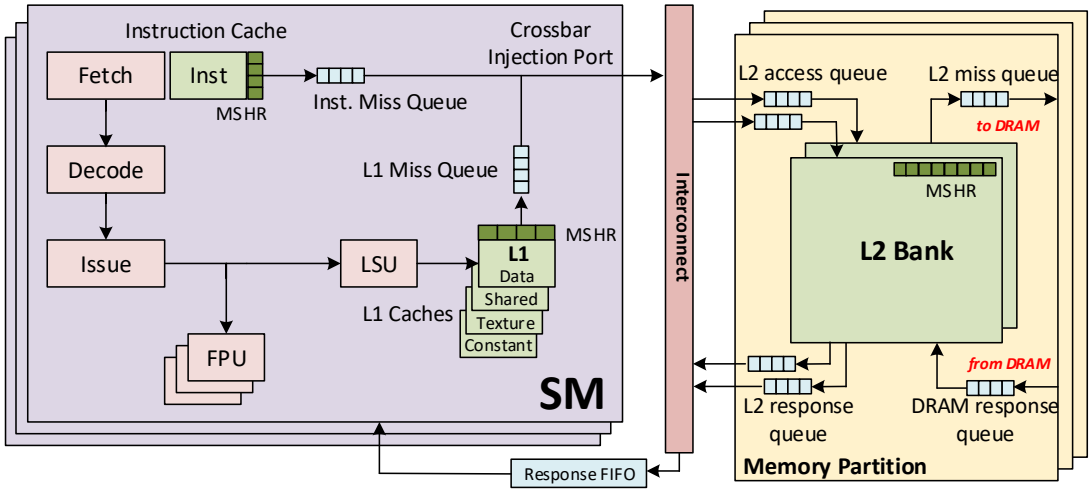


Figure 2.3: Baseline GPU architecture

and multithreading no longer hides the latency of outstanding memory operations. As a result, memory latencies appear in the critical path and directly impact performance. Memory-intensive applications are more susceptible to such a scenario due to the high frequency of memory instructions that can potentially lead to a long latency memory operation.

2.2.2 GPU Memory Hierarchy

Modern GPUs feature a memory hierarchy with impressively diverse memory units, catering to a wide spectrum of request types supported by GPUs. We now explore the different levels of the memory hierarchy and discuss some of their individual properties. More details can be found in [141, 32].

Register Files. In each SM, multi-banked register files occupy the highest level in the memory hierarchy to enable fast context switching between warps [113, 1, 111]. The register file is indexed by warp id and register id, and is used to store private data for thousands of concurrent threads under execution. Typical modern GPUs, such as Kepler, feature up to 65,536 32-bit registers per SM which amounts to 256 KB of register file per SM. This is in stark contrast to traditional multiprocessors which feature only a few hundred (or fewer) registers in the register file.

L1 Caches. The next level in the memory hierarchy is formed by the private caches within the SM. These caches (enumerated below) include read-write memories such as Shared Memory and L1 Data Cache, and read-only memories such as Constant Cache, Texture Cache and Instruction Cache.

1. *Shared Memory*: The shared memory is allocated explicitly by the programmer at thread block granularity and acts like a scratchpad for the programmer. The shared memory data is private to the thread block for which it is allocated. CUDA provides the `__shared__` memory space specifier to declare and allocate shared memory in the device code. Shared memory is heavily banked to allow simultaneous access to data for all threads of a warp in a single cycle. However, poor memory access patterns can lead to bank conflicts, thereby serializing the accesses and slowing down the warps [50, 165, 68].

2. *L1 Data Cache*: The L1 data cache is primarily used for caching global memory accesses.[2] Global memory is declared in CUDA using the `__device__` memory space specifier and can be accessed globally by all threads of a grid. The scope of global memory also extends to the host, and therefore host and device can transfer data in the global memory space. Furthermore, the L1 data caches are non-coherent and employ write-through, no-write-allocate policies for global memory accesses. This implies that a write in the global memory space is performed directly in the shared L2 cache, bypassing (on a write miss) or evicting (on a write hit) the L1 data cache. However, GPUs support a weak memory consistency model [40, 4], and therefore shared data in the remaining L1 data caches is not invalidated upon a write, potentially allowing other SMs to continue reading stale data from their respective L1 data caches. Therefore, to access the most recent data in the global memory space, the programmer or the compiler must explicitly bypass (and invalidate) the L1 data caches that can cache potentially stale data and must read directly from the L2 cache.

   In addition to global memory accesses, the L1 data cache also supports local memory accesses. Local memory is private to each thread and is declared in CUDA using the `__local__` memory space specifier. The L1 data cache employs a write-back, no-write-allocate policy for local memory accesses. Compilers use local memory space for a thread when there are not enough registers in the register file to store private thread data.

3. *Constant Cache*: The constant cache is used for caching constant memory accesses. The constant memory is declared in CUDA using the `__constant__` memory space specifier. Constant memory is read-only memory and can be accessed by all threads of a grid. Similar to global memory space, the scope

[2] In the remaining chapters, we refer to L1 data cache simply as L1 cache, unless otherwise specified.

for constant memory extends to the host, allowing host and device to transfer read-only data in the constant memory space. Constant memory is typically used when all threads of a warp read from the same memory location and the data is not expected to change over the course of execution.

4. *Texture Cache*: The texture cache is used for caching texture memory accesses. The texture memory is a read-only memory that provides a global scope and is assigned using device functions [32]. Typically, texture memory is used in scenarios when threads of a warp issue reads to contiguous memory addresses, exhibiting high spatial locality.

5. *Instruction Cache*: The instruction cache is a read-only cache accessed during the fetch stage by each warp. A warp that has a valid instruction in the instruction cache fills the instruction buffer and eventually proceeds to the decode stage. However, in case of a miss in the instruction cache, a memory request is generated and the corresponding warp is descheduled if there are no more instructions to be decoded in the instruction buffer for that warp.

L2 Cache. The private caches are backed by a shared L2 cache that adopts a write-back, write-allocate policy for L2 write requests. As shown in Figure 2.3, L2 cache is organized into multiple banks. A group of L2 banks form a *memory partition* sharing a common channel to the off-chip DRAM. The requests to the L2 cache enter the memory partition through the *L2 access queues*. Upon a read hit, the L2 cache line is read out through the data port in the L2 cache into the *L2 response queues*. On a read miss, a cache line is reserved in the L2 cache and a read to the DRAM is issued through the *L2 miss queue*. When a fill request is received from the DRAM through the *DRAM response queue*, the reserved cache line is filled and a subsequent fill request is generated for the L1 cache, if required.

Interconnection Network. Each L2 cache bank communicates with the cores through a *crossbar* interconnection network. There are separate interconnection networks for the request path (cores to L2 cache) and the response path (L2 cache to cores). The request path carries read requests and write requests, whereas the response path carries read responses and write acknowledgements. The crossbar transfers packets at the granularity of *flits*. In an uncongested memory system, L2 cache has an access latency of around 120 cycles from the L1 cache for non-texture accesses [141]. However, congestion and queueing delays due to bandwidth limitations can considerably increase the latency to access the L2 cache and DRAM. For instance, due to large cache

line size, it often takes around 4-5 cycles just to push a single cache line through the crossbar router at 32-byte flit granularity. As a result, for memory-intensive applications that present a high demand on the interconnect, the cumulative queueing delay to push multiple cache lines can be considerably high.

Graphics DRAM. The shared L2 cache is further backed by an off-chip graphics DRAM or GDDR that has an additional access latency of around 100 cycles, excluding the arbitration delays within the DRAM. Each memory partition has a memory channel with an independent DRAM command scheduler queue where the DRAM requests wait until the corresponding memory access commands are scheduled to the DRAM. DRAM scheduler employs a FR-FCFS (First-Row First-Come-First-Serve) scheduling policy to select a new request from the scheduler queue. Furthermore, each memory partition is associated with multiple DRAM chips, where each DRAM chip is organized into multiple banks. Multiple banks of a DRAM chip share a common bus that determines the peak memory bandwidth of the DRAM. Values for the above DRAM parameters used in this thesis are listed in Table 3.1 in the next chapter.

2.2.3 Memory Coalescing

Bandwidth is a critical resource in throughput-oriented architectures such as GPUs. Therefore, GPUs employ several mechanisms to reduce the bandwidth demand on the memory hierarchy. One such important technique is known as *memory coalescing*. As threads in a warp operate on a vector of data, each thread can potentially generate an independent memory request. When such threads generate highly *regular* memory accesses, *i.e.*, to consecutive words in the memory, all accesses span across a few cache lines only. In such a scenario, the load-store unit coalesces the memory requests that fall within the same cache line into a single request to the memory hierarchy. For instance, when a warp with 32 threads access 4-byte words starting from a 128-byte aligned address, all the addresses fall within a single cache line, assuming 128-byte cache line size. As a result, 32 threads of a warp generate a single, highly coalesced memory request. This phenomenon is known as memory coalescing. Such regular access patterns, combined with memory coalescing, improve memory bandwidth by allowing multiple loads to be serviced by fewer memory requests. In a different scenario, where threads access words at 128-byte offset, warp generates 32 different memory requests — one for each thread of the warp — significantly depleting memory bandwidth. Such access patterns are known to exhibit high *memory divergence* [33, 136, 60, 84]. Programmers

often try to optimize the memory access patterns to reduce memory divergence in order to maximize bandwidth [83].

After coalescing memory requests in the load-store unit, GPUs exploit another opportunity to reduce the bandwidth demand on the memory hierarchy. This is done through Miss Status Holding Registers (MSHRs) in the L1 and L2 caches. MSHRs are fully-associative arrays and are used to track outstanding memory requests. For every cache miss in progress, an MSHR entry is allocated in the corresponding cache. Subsequent accesses to the pending cache lines are merged into an existing MSHR entry that corresponds to the same address. Therefore, it eliminates redundant memory requests to the lower levels of the memory hierarchy for cache lines that are already being fetched. In other words, for every miss at the L1 and L2 caches, corresponding MSHR entries are searched for an outstanding request for the same cache line. If no existing entry is found, a new MSHR entry is created to track the new cache miss in progress. However, if an entry is found in the MSHRs, the new cache miss is merged with an existing MSHR entry that corresponds to the same cache line. Once the pending memory request is serviced, the associated MSHR entries are freed and warps associated with all the merged requests are serviced. Limiting the number of MSHR entries also limits the maximum number of memory requests that can be in-flight at any time in the memory system. Therefore, MSHRs determine the peak bandwidth demand that each level of the memory hierarchy can pose on the lower levels of the memory hierarchy.

Chapter 3

Evaluating and Mitigating Bandwidth Bottlenecks

In comparison to the traditional multiprocessors, GPUs present significantly different requirements to the memory system. This is largely because of the throughput-oriented nature of GPUs — a marked departure from the single-threaded and latency-oriented processors. In GPUs, a large number of requests are made to each level of the memory hierarchy in order to sustain the enormous parallelism in GPU cores. However, with increasing off-chip bandwidth with newer memory technologies (such as HBMs) on one side, and increasing number of streaming multiprocessors on the other side, the intermediate cache hierarchy appears to be under-provisioned to handle such high volumes of memory traffic. As a result, the bandwidth bottleneck — traditionally limited to the off-chip memory — is distributed across the entire memory hierarchy, including the cache hierarchy. This motivates us to further evaluate and understand the bandwidth bottlenecks in GPUs so that we can provision the memory hierarchy with adequate bandwidth resources and ensure a well-performing memory system.

3.1 Overview

In this chapter, we aim to characterize the severity of the bandwidth problem posed by the three levels of the memory hierarchy, *viz.*, private L1s, shared L2 and off-chip memory, and also characterize the role of their peripheral network elements such as interconnects and buffers. We show that due to bandwidth bottlenecks, there is severe congestion between the L1 and L2, as well as between the L2 and off-chip memory. As discussed in Chapter 1, such high levels of congestion lead to increased memory

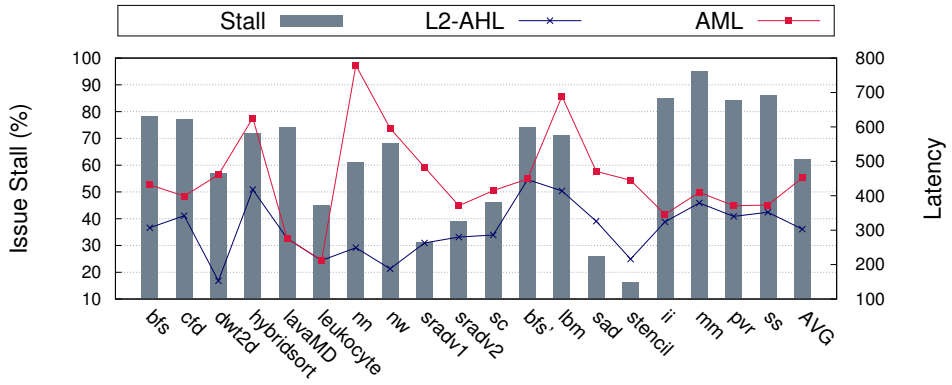


Figure 3.1: Issue-stall cycles, Average Hit Latencies to L2 (L2-AHL) and Average Memory Latencies (AML) for memory-intensive applications.

latencies, which has three major implications. Firstly, in memory-intensive applications, due to insufficient computation to mask high memory latencies, such latencies appear in the critical path of system performance. Secondly, high latencies of outstanding miss requests lead to prolonged contention of cache resources such as Miss Status Holding Registers (MSHRs) and replaceable cache lines. This effect increases the memory latencies even further as succeeding requests get serialized and have to wait for outstanding misses to complete and relinquish the resources. And thirdly, back pressure from a congested lower level further throttles the cache pipeline and prevents it from operating at peak throughput, exacerbating the bandwidth limitation in the cache hierarchy. A combination of the above factors force the cores to stall, leading to performance degradation. In Figure 3.1 we show that a set of representative memory-intensive applications, run on a simulated GTX 480 GPU, exhibit high average memory latencies (AML; 452 cycles on average) and spend a considerable fraction of application run time in a stalled state (62% on average) waiting for memory operations to complete. Additionally, high average L2 hit latencies (L2-AHL; 303 cycles on average) indicate that there is considerable congestion between the private L1 and the shared L2 cache, and therefore suggests that the high average memory latencies are due to bandwidth limitations in both the cache hierarchy and to off-chip memory.

In order to reduce congestion in the memory system, we explore several design choices at each level of the memory hierarchy and evaluate their efficacy in alleviating the bandwidth bottleneck. We conduct a design space exploration and show that solving the problem in isolation, at specific levels of the memory hierarchy, can give sub-optimal results and can even be counter-productive, only creating even more congestion

elsewhere in the memory system. For instance, we observe that to prevent throttling of the L1 cache, increasing the L1 bandwidth by increasing the number of MSHR entries to handle more outstanding misses can lead to performance degradation due to even higher congestion between L1 and L2. We verify this observation on a real GTX 480 GPU chip by increasing the core frequency, effectively increasing the L1 request rate, and observe a performance degradation (detailed discussion in Section 3.6). On the other hand, matching the increased bandwidth demand of L1 at the L2 cache significantly improves performance, and this even exceeds the performance achieved by a memory system with baseline cache hierarchy and High Bandwidth Memory (HBM) DRAM. Therefore, in order to efficiently solve the bandwidth bottleneck, we show that it is imperative to address the bandwidth demands of different memory levels in tandem and provide a synergistic solution. Additionally, we use the insights developed in this chapter to perform a cost-benefit analysis and propose efficient ways to mitigate the bandwidth bottlenecks at different levels of the memory hierarchy. Overall, this chapter expands the understanding of the bottlenecks across the GPU memory hierarchy and serves as a guide for architects and programmers to optimally scale bandwidth of the memory hierarchy and write bandwidth-sensitive programs, respectively.

Organization: The remainder of this chapter is organized as follows. Section 3.2 presents the evaluation methodology and infrastructure adopted in this chapter. Section 3.3 motivates the need for this study by showing that the bandwidth bottlenecks across the memory hierarchy in GPUs breach the latency tolerance property of GPUs by causing congestion. Section 3.4 quantifies the congestion levels across the GPU memory hierarchy and explores the architectural causes. Section 3.5 present the various design choices in the memory system to mitigate the bandwidth bottlenecks, derived from the characterization of congestion across the memory hierarchy. Section 3.6 evaluates the efficacy of mitigating the bandwidth bottlenecks through guided design space exploration. Section 3.7 identifies cost-effective configurations of the memory hierarchy and shows that synergistic scaling of L1 and L2 cache resources can reasonably match or even exceed the benefits of an HBM DRAM, leading to a performance improvement of up to 29% on average with a minimal area overhead of 1.5%. Section 3.8 discusses the relevant related work. Section 3.9 concludes the chapter by summarizing the findings and results.

Table 3.1: Baseline architecture parameters for GPGPU-Sim

Parameter	Value
Core	15 SMs, Greedy-then-oldest (GTO) scheduler
Clock frequency	Core @ 1.4 GHz; Crossbar/L2 @ 700 MHz
Threads per SM	1536
Registers per SM	32768
Shared Memory	48 KB
L1 Data Cache	16KB, 128B line, 4-way, LRU, write-evict, 32 MSHR entries, 8-entry miss queue
Interconnect	Crossbar, Fly-topology, 32B flit size
L2 Cache	768 KB, 128B line, 8-way, LRU, write-back, 12 banks, 32 MSHRs, 8-entry miss queue, 32B data port width, 8-entry request queue
DRAM	GDDR5 DRAM, Command clock 924 MHz, FR-FRCFS 384 bits net buswidth, 6 Memory Partitions, 2 DRAM chips/partition, 32-bits buswidth/chip, 8 bytes burst length, 16 DRAM banks/chip
DRAM Timing Constraints	CCD = 2, RRD=6, RCD=12, RAS=28, RP=12, RC=40, CL=12, WL=4, CDLR=5, WR=12

3.2 Background

In this section, we describe the evaluation framework used in the rest of the chapter. This includes the target architecture, simulation framework and target workloads.

3.2.1 Baseline Architecture

In this study, we consider a baseline GPU architecture similar to NVIDIA's Fermi architecture [115, 141]. Notably, as the organization of the memory hierarchy is fairly consistent across different architectures, we expect our observations to be applicable to Kepler and Maxwell as well.

3.2.2 Simulation Framework

We model a GTX 480 Fermi GPU on a cycle-accurate simulator GPGPU-Sim (v3.2.2) [10] with the baseline architectural parameters listed in Table 3.1. We use GPUWattch [98] to compute the area and power in our experiments. GPGPU-Sim shows a high performance correlation of over 97% against a Fermi GPU [141] and has been widely used in prior works [132, 1, 121, 139, 5], and therefore reliably represents our findings.

Table 3.2: List of workloads. P_∞: Speedup with infinite bandwidth memory system; P_{DRAM}: Speedup with a baseline cache hierarchy and infinite bandwidth DRAM.

#	Suite	Benchmark	Abbreviation	P_∞	P_{DRAM}
1	MapReduce	Matrix Multiplication	mm	4.90	1.01
2	Parboil	Lattice-Boltzman Method	lbm	3.40	1.87
3	MapReduce	Similarity Score	ss	3.23	1.00
4	Rodinia	Nearest Neighbour	nn	3.11	1.84
5	Rodinia	Hybrid Sort	hybridsort	3.10	1.24
6	Rodinia	Computational Fluid	cfd	3.08	1.06
7	MapReduce	Page View Rank	pvr	2.89	1.01
8	Rodinia	Breadth-First Search	bfs	2.84	1.00
9	Rodinia	Particle Potential	lavaMD	2.70	1.00
10	Rodinia	Stream Cluster	sc	2.70	1.13
11	Parboil	Breadth-First Search	bfs'	2.10	1.00
12	MapReduce	Inverted Index	ii	1.98	1.00
13	Rodinia	Speckle Reduction	sradv1	1.51	1.19
14	Rodinia	Speckle Reduction	sradv2	1.49	1.08
15	Rodinia	Needleman-Wunsch	nw	1.43	1.09
16	Parboil	PDE Solver	stencil	1.23	1.20
17	Rodinia	Wavelet Transform	dwt2d	1.20	1.14
18	Parboil	Sum of Absolute Differences	sad	1.16	1.09
19	Rodinia	Tracking Microscopy	leukocyte	1.08	1.00
			Average	2.37	1.15

3.2.3 Workloads

For the purpose of this study we use applications from three major general-purpose benchmark suites, *viz.*, Rodinia (v3.0) [27], MapReduce [57] and Parboil [146]. In Table 3.2, we list the memory-intensive benchmarks sorted by the speedup shown on an infinite bandwidth memory system (P_∞). We also show the performance improvement observed on a memory system with baseline cache hierarchy and an infinite bandwidth DRAM (P_{DRAM}). We discuss the observations regarding P_∞ and P_{DRAM} in detail in Section 3.3.2.

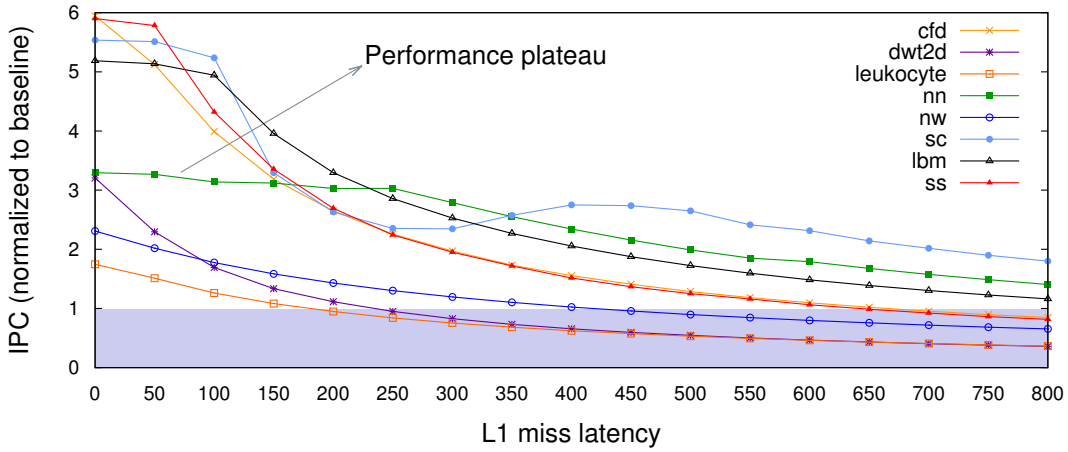


Figure 3.2: Performance variation with increasing L1 miss latency.

3.3 Motivation

In this section, we motivate the need to investigate the bandwidth bottlenecks in GPUs and discuss the potential benefits of mitigating congestion.

3.3.1 Limits of Latency Tolerance in GPUs

Multithreaded processors employ massive thread-level parallelism (TLP) to hide memory latencies. As discussed in Section 2.2.1, upon encountering an instruction that is waiting on a long latency memory operation, the corresponding warp is descheduled and an active warp (if any) is scheduled, thereby overlapping the latency of memory operation. Therefore, GPUs are usually tolerant to memory latencies. However, memory-intensive applications often run into memory misses causing all of the warps to stall due to pending memory instructions. In such a case, miss latencies get exposed due to lack of sufficient overlapping computation, and therefore lie in the critical path, adversely impacting performance.

Figure 3.2 shows the impact of memory latencies on performance, using a representative set of benchmarks from Table 3.2. In this study, we modify the memory hierarchy of the baseline architecture (described in Table 3.1) so that all the L1 miss responses are returned with a fixed and pre-determined latency that is varied in the simulator and is represented on the x-axis. The resultant performance is plotted on the y-axis which is normalized to the performance of the baseline architecture.

We observe that for most benchmarks such as *nn*, *sc* and *lbm*, the performance remains fairly tolerant to modest L1 miss latencies. This is because the cores are

able to effectively overlap such latencies with the execution of independent operations, in line with the philosophy of multithreaded architectures. However, when memory latencies are higher, there is a direct impact on performance, indicating that such high latencies lie in the critical path. For instance, IPC for *nn* reduces modestly from 3.3× to 3.03× (normalized to baseline IPC) on varying the miss latencies from 0 to 250 cycles. However, further increasing the L1 miss latencies rapidly degrades performance, reducing the IPC by 1.9× in the next 250 cycles. Furthermore, other benchmarks such as $leukocyte$ and $dwt2d$ are sensitive to even lower latencies, indicating inefficient thread-level parallelism.

We make two major observations about the *baseline memory latencies*, *i.e.*, the point on the *x*-axis where the performance curve intercepts the baseline IPC of 1× (shaded region), and therefore matches the average memory latency of the baseline architecture. First, the baseline memory latencies are significantly higher than the latencies of *performance plateau* (or peak performance) for most benchmarks. Therefore, the baseline performance is well beyond the effective operating range of latency tolerance. And second, the baseline memory latencies are also significantly higher than the ideal access latencies of L2 (120 cycles) and DRAM (additional 100 cycles via L2). This suggests that there is considerable congestion in the memory system since traversing the memory system takes significantly higher latencies than the minimum memory access latencies of L2 and DRAM. In summary, the above results indicate that there exists a significant opportunity to improve performance by reducing the latencies incurred due to congestion in the memory hierarchy.

3.3.2 Performance Impact of Reducing Congestion

In Table 3.2, we have shown the speedup obtained with an infinite bandwidth memory system (P_∞) and observed an average performance improvement of 2.37×. In such a case, L1 miss requests do not suffer any congestion-related slowdown in the memory system and only incur the minimum memory access latencies of 120 cycles to L2 (for non-texture accesses) and another 100 cycles to off-chip memory for L2 miss requests. Therefore, the speedup can be mapped to Figure 3.2 between the latency range of 120 to 220 cycles, with the average memory latency depending on the L2 miss rate. We also show the performance improvement with an infinite bandwidth DRAM appended to a baseline cache hierarchy (P_{DRAM}). In such a case, L1 miss requests suffer congestion-related slowdown only in the cache hierarchy and access the off-chip memory with

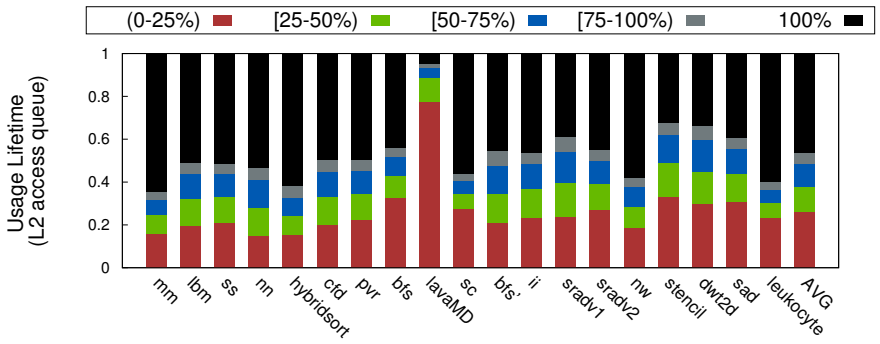


Figure 3.3: Occupancy levels in L2 access queue during the usage lifetime

a constant 100 cycle latency without incurring any congestion or timing limitations in the DRAM. In this case we notice an average performance improvement of only 1.15×, which is considerably lower than the average P_{∞}, which includes an infinite bandwidth to both caches as well as DRAM. A comparatively lower performance improvement for P_{DRAM} suggests that the existing bandwidth bottleneck in the cache hierarchy plays a crucial role in increasing the miss latencies, thereby slowing down memory-intensive applications. In the subsequent sections, we investigate the cause of such high congestion in the memory system, focusing not only on the off-chip memory but also on the cache hierarchy since it is critical for performance. We also analyse the finer implications of congestion that cause performance degradation. Using these insights, we explore the opportunities to reduce the congestion-related latencies and show how they translate to performance improvements.

3.4 Dissecting the Bandwidth Bottleneck

In a typical memory hierarchy, the bandwidth demand tapers down the memory system [148]. In principle, this is because each level filters the bandwidth demand to the lower level, and therefore the lower levels require only a fraction of bandwidth of the higher levels. However, if the bandwidth provided by the lower level is insufficient to service the bandwidth demand of the higher level, requests queue up in the memory system due to the *bandwidth skew* between the adjacent levels of the memory. This can lead to congestion in the network between the two levels, and as a consequence, requests in the higher level will have to wait for longer durations to get serviced.

In Figure 3.3 and Figure 3.4, we quantify the congestion between adjacent memory levels through an occupancy histogram of access queues to L2 and DRAM respectively.

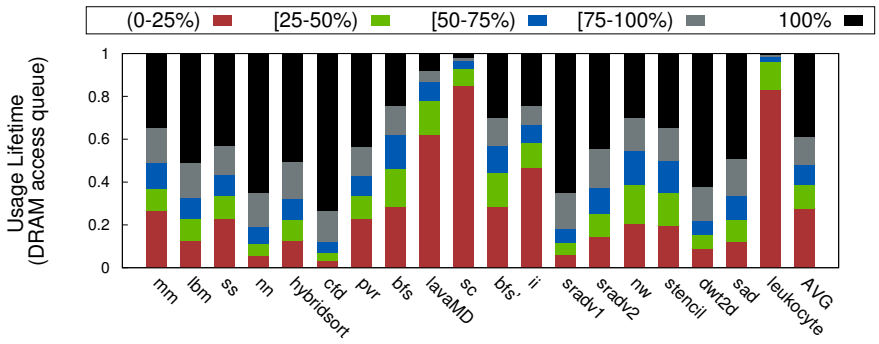


Figure 3.4: Occupancy levels in DRAM access queue during the usage lifetime

The stacked bars for each benchmark indicate the occupancy levels in the access queue, aggregated throughout the *usage lifetime* of the queue. We define usage lifetime as the time when the queues are occupied by at least one request. The occupancy histogram of the buffers between the adjacent memory levels serve as a measure of the bandwidth skew, indicating the degree of congestion between the two levels. In Figure 3.3 we note that on average, the access queues to L2 are full (indicated by the 100% occupancy bar in black) for 46% of their usage lifetime. Such high congestion aligns with the observation of high L2 access latencies. Similar to the congestion between L1 and L2, high bandwidth demand of L2 misses and low DRAM service rate causes the DRAM access queues to get full leading to congestion between the two levels. In Figure 3.4 we note that on average, DRAM access queues are full for 39% of their usage lifetime.

3.4.1 Implications of Congestion

Limited bandwidth to traverse the memory system and queuing delays due to congestion lead to high memory latencies. Such high latencies are critical to system performance and cause performance degradation (as shown in the post-plateau region in Figure 3.2). In this subsection, we delve further into the finer implications of high latencies (and congestion) and show how it leads to performance degradation. We summarize the results in Figure 3.6.

Data and Fetch Hazards: When a warp encounters an instruction that is waiting on a pending memory (or compute) operation due to a data dependency, it is descheduled and no longer participates in thread-level parallelism. This condition is known as a *data hazard*. Once the pending memory (or compute) operation completes, the data dependency is resolved and the warp is allowed to resume execution. Since floating-point operation latencies are fairly small, the majority of data hazards are caused by

pending loads [82]. When all warps are descheduled due to data hazards, which is often the case in memory-intensive applications, the core is forced to stall. In such a scenario, memory latencies contribute directly to stall cycles and govern how soon a warp can be released from a data hazard to continue execution.

Since instruction cache misses share the congested memory system with irregular data misses, high memory latencies drain the instruction buffers, thereby descheduling the warp at instruction fetch. This is known as a *fetch hazard*. High instruction cache misses can cause the fetch buffer to drain for all warps. This causes the core to stall until the instruction misses complete and the warp resumes decoding.

Structural Hazards: High miss latencies can lead to prolonged contention for limited cache resources that are used to maintain the context of outstanding miss requests. This prevents the cache from sending new miss requests to the lower level in the memory system. This condition is known as a *structural hazard*. This further adds to the miss latency since the new misses get serialized, as they have to wait for the pending requests to complete and relinquish the resources. A structural hazard can occur due to a lack of free MSHR entries in a cache to hold the context of a new miss request. Alternatively, since Fermi employs an allocate-on-miss policy for reserving new cache lines, a structural hazard can also be caused due to a lack of replaceable cache lines in a cache set if all cache lines are reserved by pending miss requests.

Memory Back Pressure: In a congested memory system, due to the inability of network queues to accept new requests, preceding queues get full. This cascading effect of congestion percolates up to the higher levels of the memory hierarchy and is known as *memory back pressure*. When memory back pressure reaches the higher level cache, it manifests as a structural hazard due to a lack of free entries in cache miss queues, and therefore prevents the cache from issuing a new miss request. For instance, back pressure from slow off-chip memory fills up the DRAM scheduler queue, in turn causing the L2 miss queues to get full. This leads to a structural hazard in the L2 cache as it cannot issue a new miss, thereby stalling the entire L2 cache pipeline. The back pressure eventually percolates up to the L1 cache and throttles core performance.

Discussion: Apart from further increasing the miss latencies, structural hazards (due to lack of cache resources or back pressure) have the following two major effects.

• *Increased hit latencies*. As structural hazards stall the cache pipeline, they prevent the succeeding requests from accessing the cache even if such requests are cache hits. This results in higher latencies for cache hits.

• *Restricted parallelism on cores*. A structural hazard in the load-store unit can

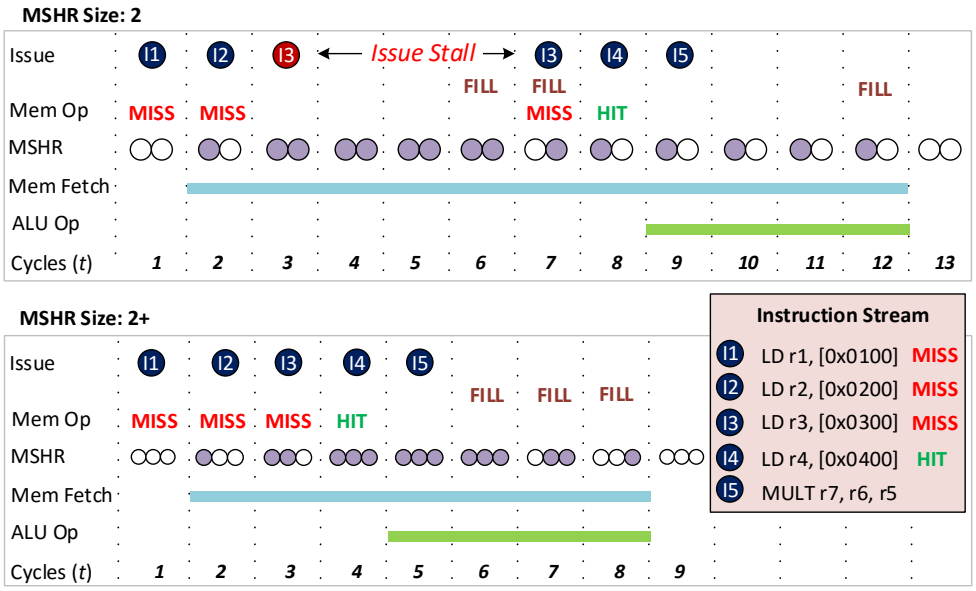


Figure 3.5: Illustrating the effects of structural hazards

cause all warps to stall when they attempt a memory instruction. This prevents the independent instructions in the instruction stream from getting issued, as the preceding memory instructions are waiting for the structural hazard to resolve, resulting in a false dependency. This serialization of memory and compute instructions prevents the core from hiding any further memory latencies, and thus performance suffers.

In Figure 3.5, we illustrate the above two scenarios with the help of an example. In the first case, we assume an MSHR with two entries, thereby allowing only two outstanding misses. Whereas in the second case, we assume a higher number of MSHRs that do not pose a structural limitation. For the sake of simplicity, we assume 6 cycles memory latency for an L1 load miss and 4 cycles for an ALU operation. In the first case, upon encountering the first two load misses, *i.e.*, I1 and I2, the MSHR gets full and can no longer accept any more misses. Since I3 is also a miss, it encounters a structural hazard and therefore stalls the L1 cache pipeline, in turn stalling the load-store unit (LSU). A succeeding cache hit in I4 needs to wait to access the L1 cache as there is a blocking I3 waiting for prior misses to relinquish the MSHR resources. Therefore, I4 gets serialized with the outstanding misses leading to a higher hit latency of I4. Additionally, a successive multiplication instruction, I5, needs to wait in the instruction queue as the previous instruction from the same warp is pending at the issue stage. This structural dependency forces the execution units to remain stalled despite an independent multiplication instruction in the instruction stream. Therefore, I4 proceeds with the hit only at $t = 8$, after the response for the first load relinquishes an MSHR

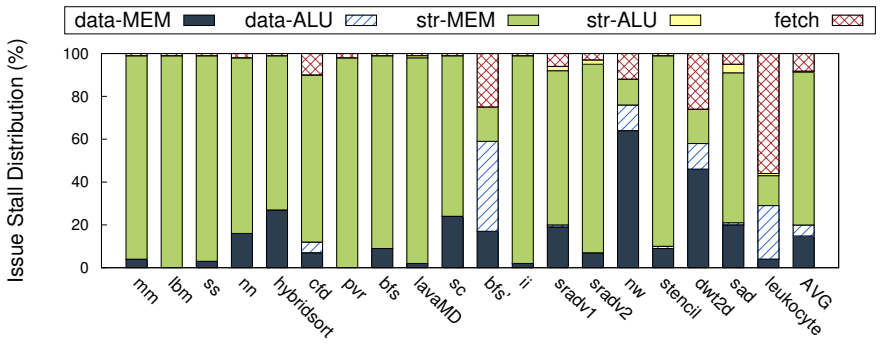


Figure 3.6: Issue-stall cycle distribution depicting stalls due to data hazards (data-MEM and data-ALU), structural hazards (str-MEM and str-ALU) and fetch hazard (fetch).

entry and unblocks the LSU. Thereafter, multiplication begins at $t = 9$ completing the execution at $t = 12$. In contrast, in the second scenario with no structural hazards, all independent instructions are issued successively. I4 results in a hit at $t = 4$ and ALUs begin computation at $t = 5$, completing the execution at $t = 8$. Note that in real systems, the miss latencies exceed hundreds of cycles, thereby magnifying the effect of such structural hazards.

Summary: In Figure 3.6, we demonstrate the distribution of the core's issue-stall cycles and attribute the cause of stall to one of the following reasons: data hazard due to a pending memory (*data-MEM*) or compute (*data-ALU*) operation; structural hazard due to resource contention in memory unit (*str-MEM*) or compute unit (*str-ALU*); and fetch hazard due to lack of instructions in the fetch buffer (*fetch*). As different warps can encounter different hazards in the same cycle, we consider a stall cycle as a data hazard when no warp can be issued due to existing data dependencies and the corresponding functional units do not pose a structural limitation for at least one warp. Similarly, a stall cycle is considered as a structural hazard when at least one warp, without any data dependencies, can be issued but is forced to stall due to resource contention in the corresponding functional units. We note that structural hazards from the memory stage form a major portion of the stalls with an average of 71% of issue-stall cycles. Data hazards due to pending memory instructions and fetch hazards contribute to 15% and 8% of issue-stall cycles on average, respectively. On average, data and structural hazards due to arithmetic units form very small portions of the issue-stall cycles, *i.e.*, 5.5% and 0.5% respectively.

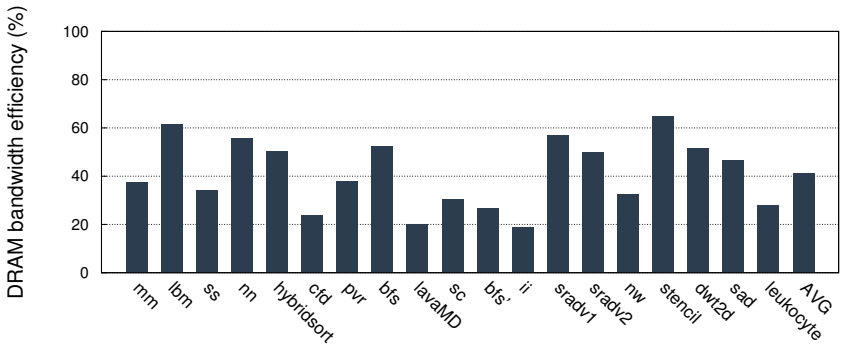


Figure 3.7: DRAM bandwidth efficiency

3.4.2 Causes of Congestion

In the previous sections, we observed that there is high congestion across the memory hierarchy due to distributed bandwidth bottlenecks, which leads to performance degradation. In order to understand the design space for mitigating congestion, we now explore the detailed causes of congestion by analysing each memory level in detail.

Off-chip Memory: Off-chip memory has been studied widely in the context of bandwidth utilization [112, 79]. DRAM timing constraints, such as *activate* and *precharge* delays, prevent DRAM from operating at peak throughput. Such constraints lead to low *bandwidth efficiency* in the DRAM, *i.e.*, the ratio of time when DRAM is transferring data on the memory bus to the time when there is at least one pending request in the DRAM scheduler queue. Therefore, a bandwidth efficiency of 100% would mean that the DRAM is always operating at peak throughput. As shown in Figure 3.7, in our experiments we observe a low average bandwidth efficiency of 41% and a maximum of 65% for *stencil*.

To improve bandwidth efficiency, several schemes have been proposed such as improving row-buffer locality [169], bank-level parallelism [112] and prioritizing read-over-writes [131]. For instance, as the overhead of opening a new row is high, reordering requests to access an already opened row leverages row-buffer locality, thereby improving DRAM throughput. Bank-level parallelism is exploited by concurrently accessing different banks in a DRAM chip, thereby masking the timing constraints across banks. Prioritizing read requests over writes minimizes the performance critical latencies of reads by trading off write latencies as they are not timing critical.

L2 Cache: Since L2 cache interacts with both DRAM at the lower level and L1 cache at the higher level, a myriad of factors can clog the L2 cache. Firstly, structural hazards due to a lack of MSHRs or non-replaceable cache lines can block the L2

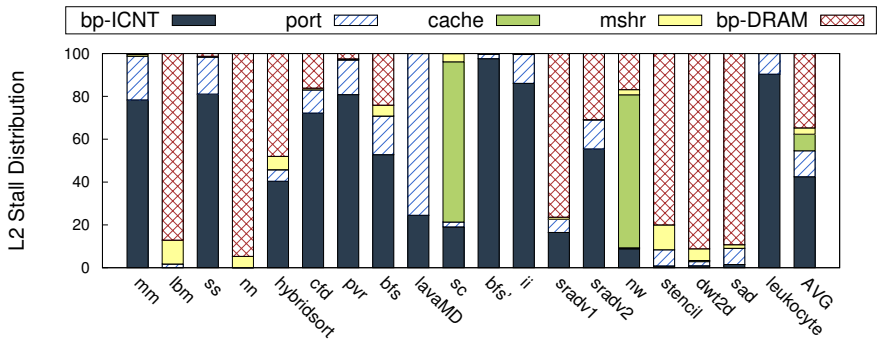


Figure 3.8: L2 stalls due to back pressure from interconnect (bp-ICNT) and DRAM (bp-DRAM) and contention on L2 data port, cache lines and MSHRs.

pipeline. Secondly, memory back pressure due to congestion in the DRAM access queues can stall the L2 miss queue, creating another structural hazard at the L2 cache. Thirdly, a busy L2 data port, due to an ongoing cache line fill from DRAM or an ongoing read of an L2 cache line, can cause port contention, forcing the subsequent L2 hits to wait before another cache line can be read. And finally, as L2 responses are injected into the crossbar at the granularity of *flits* (or network packets), it can take several cycles to inject an entire cache line. This forces the L2 responses to wait for long durations in the L2 response queue, eventually asserting back pressure on the L2.

In Figure 3.8 we quantify the L2 cache stalls due to the above factors. We note that on average, structural hazards due to a lack of MSHR entries and replaceable cache lines contribute to 3% and 8% of L2 cache stalls. Memory back pressure from DRAM contributes to 35% of total stalls, whereas L2 data port contention leads to 12% of stall cycles on average. Back pressure from L2 response queues due to slow crossbar injection rate leads to 42% of L2 stalls on average, and is thus the main cause of congestion at the L2 cache.

L1 Cache: We perform a similar analysis for L1 cache to determine the principal factors that stall the L1 cache pipeline. L1 cache can stall due to structural hazards on MSHRs as well as due to non-replaceable cache lines, similar to L2 cache. Also structural hazard due to back pressure from L2 can stall the L1 cache pipeline. In Figure 3.9 we quantify the impact of such parameters. We note that on average, MSHR and cache line contention contribute to 41% and 11% of total L1 stalls and L2 back pressure is responsible for 48% of L1 stalls. Therefore, back pressure from L2 appears as the major cause in throttling the L1 cache, followed by MSHR contention and cache contention.

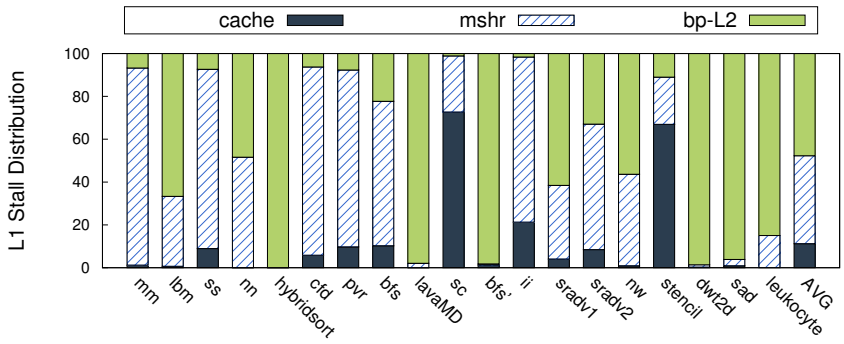


Figure 3.9: L1 stalls due to contention on cache lines and MSHRs, and back pressure from L2 cache (bp-L2).

Summary: The above discussion provides insight into the reasons behind the stalls in the memory system, and therefore serves as a guiding tool in adjusting the design to best mitigate high congestion between different levels of the memory hierarchy. We also observe the relative importance of parameters across caches. For instance, we note that the scarcity of MSHRs in L1 caches has a huge impact as they contribute to 41% of L1 stalls on average. On the other hand, MSHRs in L2 do not block the L2 cache as they contribute to only 3% of L2 stalls. We also note that back pressure contributes to a significant proportion of stall cycles at both L1 and L2 caches.

3.5 Consolidating the Design Space

In this section, we use insights from the above analysis to consolidate the design parameters that can be effective in mitigating congestion in the memory hierarchy. In the previous section, we observed that stalls at different levels of the memory hierarchy prevent caches (and cores) from operating at peak throughput. However, removing all such stalls and operating at peak throughput may not always alleviate congestion *as the peak throughput itself can be a limiting factor*. Therefore, we classify the microarchitectural parameters into the following two categories:

1) Type '=': Parameters that minimize stalls, allowing for the caches and cores to operate at peak throughput.

2) Type '+': Parameters that increase the peak throughput.

In the following subsections, we identify such parameters for the above categories and summarize our architectural design space in Table 3.3.

Table 3.3: Consolidated design space showing baseline, scaled (4×) and cost-effective configurations.

Design Parameter	Type	Baseline value	Scaled value (4×)	Cost-effective
(a) DRAM				
Scheduler queue	=	16 entries	64 entries	16 entries
DRAM Banks	=	16 banks/chip	64 banks/chip	16 banks/chip
Bus width	+	384 bits	1536 bits	384 bits
(b) L2 Cache				
L2 miss queue	=	8 entries	32 entries	32 entries
L2 response queue	=	8 entries	32 entries	32 entries
MSHR	=	32 entries	128 entries	32 entries
L2 access queue	=	8 entries	32 entries	32 entries
L2 data port	+	32 bytes	128 bytes	32 bytes
Flit size (crossbar)	+	32+32 bytes	128+128 bytes	16+48 bytes
L2 banks	+	12 banks	48 banks	12 banks
(c) L1 Cache				
L1 miss queue	=	8 entries	32 entries	32 entries
MSHR (L1D)	=	32 entries	128 entries	48 entries
Memory pipeline width	=	10	40	40

3.5.1 Off-chip Memory

The baseline architecture employs a *First-Ready First-Come-First-Serve* (FR-FCFS) scheduling policy that prioritizes accesses to an already opened DRAM row from a pool of pending requests in the scheduler queue to achieve higher row-buffer hits. To maximize the benefit of FR-FCFS scheduling, we increase the scheduler queue size and allow the DRAM to search in a larger pool of pending requests and schedule more row-buffer hits. Maximizing row-buffer hits allows the DRAM to operate closer to the peak DRAM throughput, increasing the bandwidth efficiency. In order to maximize bank-level parallelism, we increase the number of banks per DRAM chip while keeping the size of the DRAM constant. This reduces the number of rows per bank and therefore spreads the accesses to different banks, thereby increasing concurrency. Finally, to increase the peak throughput of DRAM, we increase the bus width of each DRAM chip.

3.5.2 L2 Cache

To prevent throttling of L2 cache due to back pressure from DRAM, we increase the L2 miss queue size to allow more L2 misses to be buffered in the access path to DRAM. Similarly, we increase the size of the L2 response queue to mitigate the back pressure from the response network. To reduce structural hazards due to cache resources, we increase the MSHRs. Stalls due to lack of non-replaceable cache lines can be resolved by increasing the capacity or associativity of L2 cache. However, such parameters reduce the miss traffic to the lower level, thereby altering the bandwidth demand. Since we focus on performance of the memory system given a fixed bandwidth demand, we do not alter these parameters as it leads to an unfair comparison in the context of bandwidth bottlenecks. Instead, we increase the L2 access queue size to allow more requests to be buffered at a stalled L2, avoiding back pressure to L1 cache. Therefore, all the above parameters allow L2 (and higher levels) to operate closer to the peak throughput. Finally, to increase the peak throughput of L2, we increase the L2 data port width, crossbar flit size and L2 banks. We also note that other design parameters, such as L2 and crossbar frequencies, also achieve the goal of mitigating congestion. However, we restrict ourselves to representative parameters that demonstrate the effect of increasing the L2 bandwidth.

3.5.3 L1 Cache

We reduce the impact of back pressure from L2 cache by increasing the L1 miss queue size. We also increase the MSHR entries to reduce the structural hazards. Similar to L2, we do not increase the capacity and associativity of the L1 cache to mitigate cache line contention. Instead, we increase the width of the memory pipeline on the core to allow the load-store unit to buffer more pending cache requests. The above parameters prevent the core from throttling, thereby allowing it to operate closer to the peak throughput.

3.6 Design Space Exploration

In this section, we evaluate the design space by scaling the bandwidth of different levels of the memory hierarchy through the architectural knobs listed in Table 3.3. As a typical HBM [56] provides up to 4$\times$ bandwidth compared to GDDR5 DRAM, we evaluate similar factor of scaling in other levels of the memory.

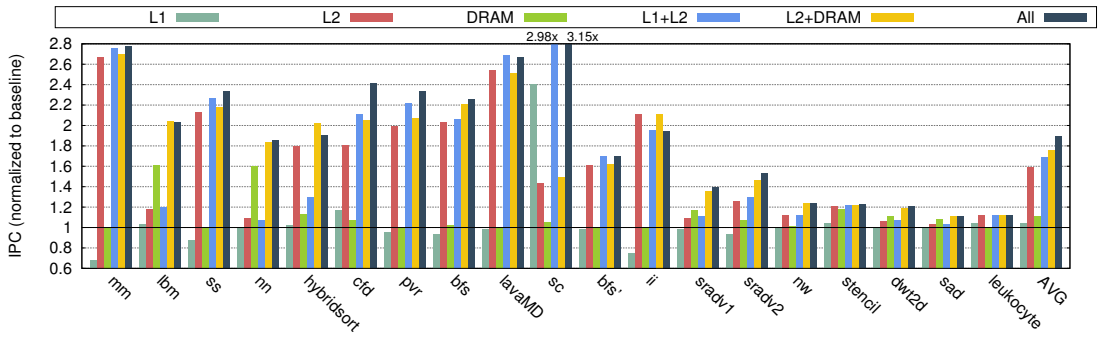


Figure 3.10: IPC gain with 4× design-point scaling of bandwidth resources in L1, L2, DRAM and synergistically across different levels.

3.6.1 Results

In Figure 3.10, we demonstrate the results obtained by scaling the design parameters by a factor of 4×. We begin by discussing the performance improvement from increasing the bandwidth in independent levels of the memory hierarchy. Later, we discuss the combined effects of increasing the bandwidth across adjacent memory levels, followed by scaling the bandwidth across the entire memory hierarchy.

L1 Cache: On increasing the L1 resources, we see an average performance improvement of 4%. We observe the maximum speedup of 240% for *sc* followed by a speedup of 16% for cfd. The reason for the observed speedup lies in the fact that increased resources reduce the structural hazards on L1 cache. This results in better overlap of memory operations with computation and lower latencies of cache hits, as illustrated in Figure 3.5.

On the other hand, for some other benchmarks, the performance drops when the L1 resources are increased. For instance, *mm* and *ii* suffer a slowdown of 33% and 25% respectively. This is because, although increasing the L1 resources allows the L1 cache to operate at peak throughput, at the same time it also leads to higher congestion between L1 and L2, as the increased bandwidth demand of L1 is not matched by the bandwidth provided by L2. Since higher congestion causes greater interleaving of requests from different cores, requests from the same core (and therefore same warps) get more sparse in the memory system, thereby delaying the *tail request* of a warp. Since a core can resume execution only on receiving all the memory requests generated by a warp, it causes significantly higher stalls as none of the cores can resume execution any earlier than baseline. Additionally, we also notice a significantly higher L2 miss rate for applications showing slowdown. For instance, the L2 miss rate increases from 16%

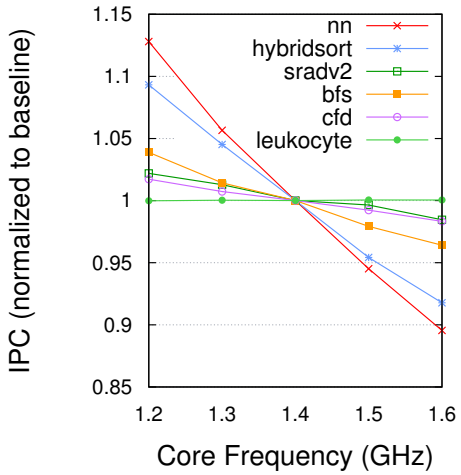


Figure 3.11: Core frequency variation on real GTX 480 GPU.

to 58% for *mm* and from 15% to 62% for *ii*. This is also due to higher interleaving of request streams from different cores that exhibit low inter-core locality, thereby causing cache thrashing and destroying the intra-core locality in the L2 cache.

We verify the above behaviour on a real GTX 480 GPU by increasing the core frequency for representative benchmarks and note a performance degradation of up to 10%, as shown in Figure 3.11. Increasing the core frequency is analogous to increasing the L1 cache resources as it increases the request rate (or bandwidth demand) from L1 to L2. Interestingly, performance improves on reducing the core frequency as the reduced bandwidth demand by L1 resonates well with the bandwidth offered by L2.

L2 Cache: By scaling the L2 cache resources, we observe an average performance improvement of 59%. We observe the maximum speedup of 266% for *mm*, which is also the most bandwidth-sensitive application. A significant performance improvement by scaling the L2 parameters signifies the criticality of the L2 bandwidth to the overall system performance.

Off-chip Memory: Upon increasing the DRAM bandwidth, we observe an average performance improvement of 11%. We observe the maximum speedup of 61% for *lbm* followed by a speedup of 60% for *nn*. Note that the improved DRAM bandwidth matches the bandwidth offered by High Bandwidth Memory (HBM) and is representative of HBM performance. We note that the average improvement is in close proximity to the performance improvement of 15% obtained on a memory system with baseline cache hierarchy and an infinite bandwidth DRAM (average P_{DRAM}). However, it is considerably less than the performance improvement achieved on increasing the L2 cache bandwidth.

L1 and L2 Cache: Upon synergistically increasing the bandwidth of the cache hierarchy, we observe an average performance improvement of 69%. We note that it is higher than the sum of gains obtained by improving the bandwidth in both cache levels independently, *i.e.*, 4% from L1 and 59% from L2. We observe in *mm* that even though increasing the L1 bandwidth alone resulted in performance degradation of 33%, increasing the L1 bandwidth with L2 results in a performance improvement of 276%, which is even higher than the 266% obtained by increasing the L2 bandwidth alone. A similar effect is seen in *ss*. We can therefore conclude that despite a slowdown on increasing the structural resources at L1, synergistic scaling of L1 and L2 results in a much higher performance improvement, which is greater than the standalone improvement of L2. We also observe that the average speedup by mitigating the bandwidth bottleneck in the cache hierarchy (69%) is significantly better than the speedup obtained by a memory system with baseline cache hierarchy and an HBM DRAM (11%).

We note an exception for *ii*, where combined scaling of L1 and L2 led to a lower speedup when compared to standalone scaling of L2 cache. However, we verify in our experiments that on further increasing the L2 bandwidth, synergistic scaling starts giving better results. This indicates that for *ii*, the increased L2 bandwidth in Figure 3.10 is not yet sufficient for the increased bandwidth demand of L1.

L2 and Off-chip Memory: We observe an average performance improvement of 76% upon increasing the bandwidth at both L2 and DRAM. It is worth noting this is in close proximity to the average speedup obtained by synergistically scaling the L1 and L2 bandwidth (69%).

All Memory Levels: We observe an average performance improvement of 90% on increasing the bandwidth of the cache hierarchy as well as the off-chip memory.

3.6.2 Summary

We conducted a limited design space exploration on architectural parameters relevant to the memory bandwidth in GPUs. We observed an average speedup of 4%, 59% and 11% on increasing the bandwidth of L1, L2 and DRAM alone. We further observed an average speedup of 69% and 76% on increasing the combined bandwidth of L1-L2 and L2-DRAM. Finally, we observed an average speedup of 90% on increasing the bandwidth of the entire memory system. Therefore, we demonstrate the criticality of cache hierarchy in mitigating congestion. We also demonstrate that synergistic scaling

yields better results than increasing the bandwidth of the memory levels independently. And finally, we show that mitigating congestion in the cache hierarchy exceeds the benefit obtained by a memory system with HBM DRAM.

3.7 Cost-Benefit Analysis

In Section 3.5, we classified the architectural design space into two categories: Type '=' and Type '+'. Later, we evaluated the effect of scaling these parameters by a factor of 4×. However, such scaling across all parameters is typically not practical due to cost overheads. Therefore, we qualitatively analyse the cost versus benefit associated with the parameters in the design space and arrive at a cost-effective configuration to scale the bandwidth across the memory hierarchy. We summarize the cost-effective configuration parameters in Table 3.3.

3.7.1 Cost-effective Design Space

Type '=' parameters listed in Table 3.3 typically include buffers and MSHRs, and enable the memory levels to operate closer to the peak throughput. Buffers are simple structures and present minimal overhead in scaling. However, MSHRs are fully associative arrays and indexing high number of requests can be expensive. Since we have already observed in Figure 3.8 that L2 seldom stalls due to MSHR contention, we consider increasing MSHRs only in the L1 cache.

Type '+' parameters in the cache hierarchy such as crossbar flit size, L2 data port width and L2 banks are more complex than simple buffers and MSHRs, and therefore incur considerable cost in scaling. As shown in Figure 3.8, L2 data port only contributes to 12% of total L2 stalls on average. Due to its low contribution to the overall L2 stalls, we do not consider it for scaling. On the other hand, back pressure from interconnection network contributes to 42% of L2 stalls on average. While both L2 banks and flit size improve interconnect bandwidth and resolve such stalls, we do not consider increasing the L2 banks. This is because each L2 bank has an independent port to the crossbar; therefore, increasing the L2 banks would lead to higher number of routers in the crossbar, in turn increasing the router area. In addition, router at the cores would need to arbitrate over higher number of destinations, increasing the energy demands of the crossbar. Instead, we only increase the flit size of the crossbar as it increases the point-to-point bandwidth without significantly increasing the router area or arbitration energy.

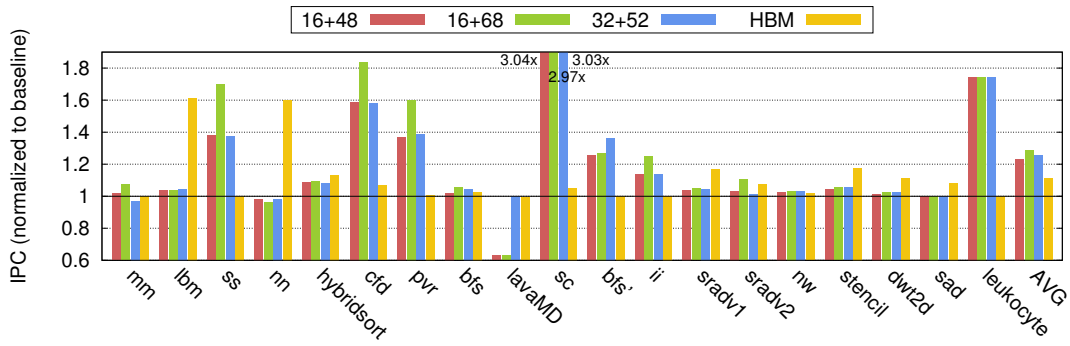


Figure 3.12: Performance gain with cost-effective configurations in order of increasing or equal cost overheads, normalized to the baseline architecture.

3.7.2 Asymmetric Crossbar

The baseline crossbar offers a uniform flit size of 32 bytes for all nodes between core-to-L2 as well as L2-to-core. However, the bandwidth demand of the reply network (L2-to-core) is higher than that of the request network (core-to-L2). This is because the majority of request packets are load requests that amount to only 8 byte packets, whereas the majority of reply packets are load responses that amount to 128 byte cache lines. Although write requests in the request network present a higher bandwidth demand, such requests are relatively infrequent and the latency for such requests is not in the critical path of system performance. Therefore, we consider an asymmetric crossbar with lower request bandwidth (16 bytes) and higher reply bandwidth (48 bytes), henceforth referred to as the *16+48* crossbar configuration. Note that we do not increase the net area of the crossbar as the total number of point-to-point wires in the *16+48* crossbar are same as the baseline *32+32* crossbar. We also discuss other crossbar configurations such as *16+68* and *32+52* with minor cost overheads over the baseline architecture.

3.7.3 Results with Cost-effective Configuration

As shown in Figure 3.12, for the *16+48* cost-effective configuration summarized in Table 3.3, we observe an average performance improvement of 23.4%. It exceeds the average performance improvement of 11% with HBM. We note an exception for *lavaMD* which shows a performance drop of 37%. This is because *lavaMD* is limited at L1 by the L2 back pressure (Figure 3.9) which gets aggravated due to reduced flit size in the request network. Even increasing the interconnect reply bandwidth does not cause

much benefit as it is limited at L2 by the data port width (Figure 3.8). Additionally, in our experiments we note that a standalone asymmetric crossbar without scaling other L1 and L2 parameters result in a lower speedup of 15.5%, thus emphasizing the importance of synergistic scaling.

We also evaluate *16+68* and *32+52* cost-effective configurations and observe a performance improvement of 29% and 25.7% respectively. While both the above crossbar configurations have equal point-to-point connections in total, we notice higher reward in investing more bandwidth in the reply network due to its higher bandwidth demand.

Overhead: We use GPUWattch [98] to estimate the area of our proposed architecture. We first compute the additional storage required in the cost-effective configuration for buffers and MSHRs. We assume each buffer entry to be 128 byte wide, while each miss queue and MSHR entry to be 8 byte wide. This results in a net storage overhead of 94 KB and amounts to an area overhead of 6.9 mm^2 at 40 nm technology, computed using existing values in GPUWattch. This amounts to an overall increase in the die area by 0.98% with respect to the baseline processor area of 700 mm^2. We do not report power overhead as it is minimal and within the margin of error of the simulator.

The baseline *32+32* interconnection network occupies a total area of 27 mm^2, while the wires contribute to 11.6 mm^2. Therefore, on increasing the point-to-point connections by 20 bytes in *16+68* and *32+52* crossbar, we incur an additional overhead of 3.62 mm^2. Therefore, along with overhead of buffers and MSHRs, the above two configurations result in a net area overhead of around 1.5%.

3.8 Related Work

In this section, we discuss prior work related to the proposals discussed in this chapter.

3.8.1 Cache Bypassing and Request Reordering

Several prior schemes have been proposed for GPUs to reduce the performance impact of bandwidth bottlenecks and high memory access latencies. Some of these schemes pertain to bypassing congested memory resources, while others include reordering memory requests to maximize cache performance and minimize bandwidth demand. Jia *et al.* [70] proposed a memory request and prioritization buffer (MRPB) where they bypass the L1 cache when high contention is detected due to memory back pressure or

structural hazards at the L1 cache. This helps prevent stalling of the memory pipeline and reduces the impact of congestion and bandwidth bottlenecks on performance. They also propose reordering of memory requests from different warps and thread blocks to improve the locality in the access stream. Through such reordering, requests from the same source, such as warp or thread block, are grouped together to minimize cache thrashing as they are expected to have higher locality. The new order of requests allows for more cache-friendly access patterns, thereby improving cache performance. Li *et al.* [99] proposed a more proactive way of cache bypassing, rather than reactively bypassing the cache upon detecting congestion. In order to do so, they detect the locality in the access stream and allow the L1 cache to be used only for requests with high reuse and short reuse distances. The remaining accesses bypass the L1 cache proactively, thereby reducing cache thrashing and circumventing congestion. In their scheme, the reuse characteristics are maintained and preserved by decoupled tag and data arrays in the L1 cache. Xie *et al.* [161] proposed a coordinated static and dynamic scheme to perform cache bypassing. At compile time, they determine the loads that demonstrate high locality and mark such loads to be allocated in the cache. On the other hand, loads with poor locality are marked to bypass the cache. At runtime, the hardware uses the above prior knowledge about locality characteristics to bypass or allocate the cache. However, loads with moderate locality characteristics are selectively bypassed for a fraction of thread blocks, where this balance is determined dynamically on the basis of cache contention and resource congestion. Furthermore, Chen *et al.* [28] proposed a hybrid scheme with coordinated cache bypassing and warp throttling (CBWT). In their scheme, they enable cache bypassing on detecting high contention for cache resources, thereby protecting cache lines with high reuse to reduce cache thrashing. In addition, they monitor the NoC latencies and enable dynamic warp throttling when high congestion in the memory system is observed. They do so by iteratively altering the number of warps by hill climbing to optimize NoC latencies, thereby regulating congestion. More recently, Lee and Wu [94] proposed *Ctrl-C*, an instruction-based scheme that detects the reuse characteristics at per-instruction granularity and alters the bypass aggressiveness accordingly for each instruction. Similarly, Koo *et al.* [86] proposed *APCM*, an instruction-based scheme to not only bypass, but also to protect cache lines using instruction locality characteristics. In their scheme, they detect streaming access patterns in a single warp and use that to bypass the L1 cache for similar instructions from the remaining warps. In addition, they detect instruction sequences that show high tendency for reuse, such as loops, and protect high locality

cache lines in the L1 cache only for the duration of these instruction sequences.

Sethia *et al.* [139] recognized memory back pressure as a significant problem in GPUs. In their scheme, they allow memory requests to continue accessing the L1 cache despite a stalled LSU. This helps in resolving false dependencies and serialization caused by structural hazards and memory back pressure, thereby allowing the hidden hit-under-miss requests to access the L1 cache without explicitly waiting for the back pressure to resolve. In turn, this helps in reducing the hit latencies and improving hit rate in the L1 cache. They also proposed a scheduling policy where memory requests from a single warp are issued to the memory system, instead of issuing memory requests from all warps. This helps in enabling at least one warp to be serviced quickly so that it can resume execution. On similar lines, Kim *et al.* [82] explored an opportunity where independent instructions in a warp that follow the dependent instruction can be executed while the warp is waiting for outstanding memory accesses to complete. This is in contrast to baseline scheme where warps are descheduled on encountering the first dependent instruction that is waiting on a pending operation. Subsequently, they proposed pre-execution of such newly extracted independent instructions in the warp. The output of such instructions is stored temporarily in renamed physical registers. This pre-executed output is then utilized in the actual execution sequence, thereby preventing write-after-write and write-after-read hazards that might occur due to reordering. This improves the latency hiding ability of the GPU due to higher opportunities to overlap memory latencies.

In summary, the above cache bypassing and request reordering schemes allow GPUs to circumvent the stalls arising due to bandwidth bottlenecks or reduce the bandwidth demand by improving cache performance. In contrast, our focus in this chapter is to mitigate the stalls (not circumvent the stalls) by improving the bandwidth resources, for a given bandwidth demand and cache performance (not reducing the bandwidth demand). Given the huge bandwidth bottleneck in current GPUs, we expect above techniques to be complimentary to our proposed cost-effective design space. Several other proposals related to cache management and warp scheduling aim to maximize cache utilization to reduce the bandwidth demand by capturing locality in the caches and reducing thrashing. These proposals are presented and discussed in the subsequent chapters (see Section 4.7 and Section 5.8).

3.8.2 On-chip Networks in GPUs

In the field of interconnection networks, several prior studies have recognized the importance of NoC bandwidth in GPUs. Bakhoda *et al.* [11] proposed a *checkerboard* organization of the mesh network where routers alternate between full-routers and half-routers. A full-router provides full connectivity between all ports in a 2D mesh, whereas a half-router limits the connectivity, thereby simplifying router design to exploit many-to-few and few-to-many traffic pattern in GPUs. Kim *et al.* [80] proposed *DA2mesh*, a cost-effective design to reduce congestion by increasing the NoC frequency in GPUs, while maintaining the same channel width. They assume a mesh topology for the interconnection network in their study. They also identify the criticality of response network with respect to bandwidth demand. Subsequently, they propose a heterogeneous NoC which consists of non-identical request and response networks. For the request path, they employ the baseline 2D mesh, whereas for the response path, they propose a direct all-to-all network overlaid on mesh with direct connections between memory controllers and SMs. Direct connections reduce arbitration, thereby simplifying the router and relaxing critical path requirements, allowing higher router frequency. On similar lines, Mishra *et al.* [110] identified that a single monolithic interconnection network is not suitable for CPU applications with divergent needs. Therefore, they proposed heterogeneous and multiple interconnection networks, individually customized for latency-sensitive and bandwidth-sensitive applications. The bandwidth-sensitive design provides a wider but low frequency network, whereas latency-sensitive design provides a narrower but high frequency network. While this chapter focusses on different bandwidth demands for request and response path within the *same* application, the above work is similar in the fact that it identifies the need for heterogeneity in designing interconnection networks based on divergent bandwidth demands. Ziabari *et al.* [175] evaluated the design space across different network topologies for GPU workloads. They proposed independent and parallel networks for reply and response, while also eliminating L1-to-L1 connections for energy efficiency — this is used as a baseline in this chapter. In addition, they also recognized the lower bandwidth demand of the request network in comparison to the response network. Therefore, they proposed an asymmetric NoC architecture by shrinking the width of the request network to improve energy efficiency. In our work, we share the above observation about asymmetric bandwidth requirements for request and response networks. However, we perform a design space exploration across the entire memory hierarchy, going

beyond interconnection networks. Thereafter, we motivate a synergistic improvement of bandwidth across the memory hierarchy, *i.e.,* an asymmetric interconnection network alongside improvements to *other bandwidth resources across the memory hierarchy as well*, such as MSHRs and buffers. Zhao *et al.* [173] also exploited the asymmetry in bandwidth requirements for request and response networks. While using a mesh network as baseline, they proposed *cfNOC*, a conflict-free design consisting of column-independent token mesh for the request network. This leads to simplified router design by removing input buffers, VC allocator and switch allocator from baseline router design in the request network, thereby reducing the hardware cost. In a subsequent work, Zhao *et al.* [172] demonstrated that the response network is often congested and leads to high queuing latency for GPGPU workloads. They proposed *HRCnet*, which consisted of a ring-chain network for the response path. They show that the proposed ring network allows for higher channel width, without increasing the overall bisection bandwidth. Therefore, HRCnet results in lower queueing latencies and scales gracefully under high injection rates, providing a better alternative to 2D mesh networks in GPUs. In contrast to the above prior techniques, we motivate a synergistic treatment of the bandwidth bottleneck in GPUs.

3.8.3 Design Space Exploration

In prior work, analytical models have been proposed to explore the architectural design space and construct balanced memory hierarchies with respect to bandwidth. Sun *et al.* [148] proposed *Moguls*, an analytical model to optimize the memory hierarchy design by quickly exploring the design space suitable for an application. The model computes the optimum bandwidth, cache levels and cache capacity to match the bandwidth demand of an application. Gulur *et al.* [52] proposed *ANATOMY*, a queuing theory based analytical model to study the off-chip memory design space in multicores. In their model, they first capture the key workload characteristics, such as memory access locality and bank-level parallelism, that are relevant to model the memory system performance. Thereafter, these workload characteristics are evaluated with a simple queuing model of the off-chip memory across different configurations and design choices to estimate memory system performance. In contrast to the above proposals, we investigate the finer parameters that lead to congestion in the existing memory hierarchy. Therefore, we go beyond off-chip memory by including parameters such as MSHRs, interconnect bandwidth, $etc.$, that lead to congestion in a given memory hierarchy. In

addition, we mitigate congestion without changing the cache capacity and the number of memory levels, *i.e.*, maintaining a fixed bandwidth demand. Therefore, proposals that alter the bandwidth demand are orthogonal to our work.

O'Neil and Burtscher [121] evaluated the performance bottlenecks in GPUs due to control flow divergence and memory access irregularity. They also examined the impact of cache size, latency and bandwidth, as well as main memory latency and bandwidth. They observed a trend that GPUs are more sensitive to L2 bandwidth than DRAM bandwidth for irregular workloads, an observation also made in our work. However, in our work, we perform a more fine-grained analysis of the microarchitectural bottlenecks. Additionally, we go beyond examining the bandwidth trends in the memory hierarchy, and propose a *cost-effective* design space based on the insights about stalls in the memory hierarchy. Alsop *et al.* [5] proposed *GSI*, a GPU stall inspector to identify source of stalls in tightly coupled heterogeneous CPU-GPU architectures, in contrast to discrete GPUs as done in our work.

3.9 Conclusion

In this chapter, we evaluate the bandwidth limitations posed by the memory hierarchy in GPUs. We observe that the bandwidth bottlenecks are distributed across the entire memory hierarchy and are not just limited to the off-chip memory. We also observe that bandwidth bottlenecks lead to high congestion in the memory hierarchy, in turn leading to high latencies that appear in the critical path. We characterize the stalls across the memory hierarchy and isolate the causes of congestion at each memory level.

After a rigorous characterization of the bandwidth bottlenecks, we identify the key architectural parameters across the memory hierarchy that prevent the different levels from operating at peak throughput or inherently limit the peak throughput. Using these architectural knobs, we perform a design space exploration and demonstrate that increasing bandwidth in isolation at specific levels of the memory hierarchy can be sub-optimal, and can even lead to performance degradation. We also demonstrate that the performance improvement obtained by synergistically improving the bandwidth of the cache hierarchy surpasses the speedup achieved by a memory system with baseline cache hierarchy and HBM DRAM. Finally, using the insights developed in this work, we perform a cost-benefit analysis and identify cost-effective configurations of the memory hierarchy to best mitigate the bandwidth bottlenecks. Our cost-effective configuration comprises an asymmetric crossbar, alongside other architectural optimizations that

allow L1 and L2 to operate closer to the peak throughput. We show that our final configuration achieves a performance improvement of 29% on average with a minimal area overhead of 1.5%. In summary, this chapter provides a methodology to improve the bandwidth of the memory hierarchy by quantifying the bandwidth bottlenecks and investing resources at the most potent bottlenecks in a cost-effective manner.

Chapter 4

Cooperative Caching for GPUs

GPUs are no longer perceived as accelerators solely for graphic workloads, and now cater to a much broader spectrum of applications. The massive compute power of modern GPUs and recent innovations in their architecture [115, 116] have helped unleash the latent potential of several non-graphical applications. The cache hierarchy — adopted from traditional CPUs — is one such innovation to capture the locality needs of upcoming applications and advance the pervasiveness of GPUs. However, the cache management policies that are suitable for CPUs may not be suitable for GPUs, evident from the high cache miss rates seen on many GPUs. One such inefficiency in current GPU cache management policies is the repeated access to the shared L2 cache from different L1 caches for same data, arising due to inter-core data reuse. Such a policy is a common occurrence in CPUs and is generally benign for performance when L1 miss rates are low, as it does not excessively deplete the memory bandwidth. In GPUs, however, such a policy is corrosive to the overall performance due to high L1 miss rates. As a result, for memory-intensive applications, where performance is constrained by memory bandwidth, such a policy aggravates the bandwidth issue by repeated memory requests for data already cached elsewhere at the same level in the memory hierarchy. Therefore, it is critical to address the inefficiencies of the existing cache management policies in GPUs to ensure effective usage of scarce bandwidth resources.

4.1 Overview

In this chapter, we aim to reduce the congestion across the memory hierarchy in GPUs by addressing the bandwidth aspect of inefficient cache management. The presence of such inefficiency in cache management is indicated by the high L1 cache miss rate, as

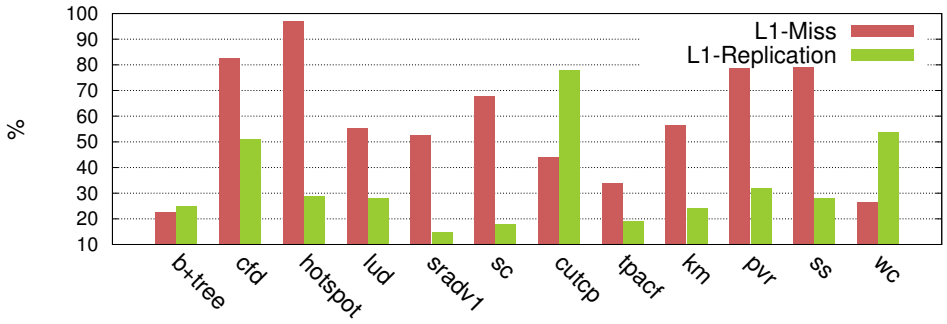


Figure 4.1: (a) *L1-Miss*: L1 cache miss rates (b) *L1-Replication*: Percentage of L1 misses cached in remote L1 caches.

shown in Figure 4.1 for a variety of general-purpose applications run on a simulated GTX 480 GPU. As discussed in Chapter 3, high L1 miss rates lead to increased pressure on L2 bandwidth, thereby increasing memory access latencies due to congestion in the L2 access path. In our experiments (discussed later in Section 4.5), we observe that due to congestion in the L1-L2 interconnect and L2 access queues, L2 accesses take up to 2-3$\times$ more cycles compared to the minimum access latency of L2. Due to lack of sufficient independent operations in memory-intensive applications to overlap such high memory access latencies, increased latencies to the lower level get exposed and appear in the critical path, reducing system performance. In Chapter 3, we identified the critical bandwidth bottlenecks across the memory hierarchy in GPUs and explored the architectural parameters that can be scaled to alleviate the bandwidth bottlenecks. However, given the large magnitude of the bandwidth bottleneck, indicated in Table 3.2 by the potential speedup of 2.37$\times$ shown in absence of any congestion, the problem requires redressal at multiple fronts. Therefore, in this chapter, we aim to reduce the congestion in the memory hierarchy by improving the aggregate caching efficiency of the L1 cache, thereby reducing the demand on the shared L2 bandwidth.

Observation: In streaming applications, cores work on independent data with little or no overlap in the working dataset. However, in general-purpose applications we observe a considerable potential for data reuse across different cores. Figure 4.1 shows that a significant percentage of miss requests generated by L1s is for data already present on a *non-local (or remote) L1 cache*. If we can exploit this reuse within the L1s, duplicate requests to the shared L2 can be potentially eliminated. This would result in reduced congestion and faster lower level access for the remaining requests.

Proposal: In this chapter, we propose a Cooperative Caching Network (CCN) for

L1 caches in GPUs to improve the aggregate efficiency of the L1 cache hierarchy in filtering requests to the L2 cache. In our proposed scheme, we connect the private L1 caches in a lightweight ring network to facilitate communication of reusable data among the L1 caches. In doing so, we reduce the average memory access latency due to the following two reasons. Firstly, a fraction of L1 load misses, with reusable data cached on remote L1s, can now completely bypass the high latency access path to L2; they are instead serviced by the CCN with significantly lower latencies (42 cycles on average based on our experiments) as compared to the L2 roundtrip access latencies, or simply L2 access latencies (which is $\sim$300 cycles due to congestion). Secondly, cooperatively sharing reusable data within the L1 caches via the CCN reduces the traffic to L2 cache. This relieves the pressure on the interconnect as well as on the L2 access queues, thereby reducing the L2 access latencies (by 78 cycles on average). Thus, CCN provides a faster access to L2 for miss requests that do not find a sharer in the CCN.

In effect, our proposed architecture services a portion of L1 misses collaboratively within the L1 caches with much lower latencies than the L2 access latency. This leads to less congestion in the L2 access path, thereby accelerating the response from memory for requests that do not find a reusable copy in remote L1 caches. However, in the absence of reuse (such as in streaming applications), unsuccessful probes in the CCN adds an additional overhead to the L1 load misses. In such cases, due to no reduction in congestion, the CCN overhead is not ameliorated, and this results in an overall performance penalty. Therefore, in our final scheme we propose CCN-RT, a Cooperative Caching Network with Request Throttling. It dynamically adapts to the coarse-grained reuse patterns exhibited by the application, bypassing the CCN when there is little or no reuse.

Organization: The remainder of the chapter is organized as follows. Section 4.2 provides an overview of the baseline architecture for our study and characterizes the workloads. Section 4.3 investigates the reuse patterns and provides a fresh insight into the inter-core reuse patterns within the GPUs by profiling the communication characteristics over a diverse range of GPGPU applications. Furthermore, we assess the efficacy of cooperative caching in GPUs. Section 4.4 presents CCN, a Cooperative Caching Network for L1 caches in GPUs that is cognizant of the inter-core reuse. Section 4.5 evaluates the architecture and proposed optimizations to our baseline proposal. We show that CCN reduces the overall bandwidth demand to L2 cache by servicing reusable requests via the CCN, boosting performance for memory-intensive applications that show high levels of sharing across L1s. With our final proposal

Table 4.1: Baseline architectural parameters for GPGPU-Sim

Parameter	Value
Core	15 SMs, Greedy-then-oldest (GTO) scheduler
Clock frequency	Core @ 1.4 GHz; Interconnect/L2 @ 700 MHz
Threads per SM	1536
Warp width	32
SIMD lane width	32
Registers per SM	32768
Shared Memory	48 KB
L1 Data Cache	16KB, 128 byte line, 4-way, LRU, write-through, no-write-allocate
L2 Cache	768 KB, 128 byte line, 8-way, LRU, write-back, 12 banks
DRAM	GDDR5 DRAM, 6 channel, 64-bits per channel, 924 MHz

CCN-RT, we show an average performance gain of 14.7% for applications that exhibit reuse, while being benign to applications with no reuse. We also reduce the average memory latency by 24%, L1 to L2 traffic by 29% and core stall cycles by 26%. Our proposal incurs nominal area and energy overheads of 1.3% and 2.5% respectively. Section 4.6 compares CCN with prior techniques. Section 4.7 discusses the related work and positions our findings in the current state-of-the-art. Section 4.8 concludes the chapter by summarizing the findings and contributions of this work.

4.2 Background

In this section, we outline the necessary background for this chapter that includes the baseline architecture and the evaluated workloads.

4.2.1 Baseline Architecture

In this study, we consider a baseline similar to NVIDIA's Fermi architecture. Our baseline GPU consists of 15 SMs, each with a 32 lane SIMD unit. As discussed in Chapter 2, each core consists of a private L1 data cache, shared memory (scratchpad) and read-only instruction, texture and constant caches. Private caches of a core are backed by a shared L2 cache that has an access latency of 120 cycles for non-texture accesses in an uncongested memory system. The L1 data caches are non-coherent and employ *write-through, no-write-allocate* policies. The baseline parameters are summarized in Table 4.1.

Table 4.2: Benchmark characterization: (a) PerfX - speedup with perfect memory (b) μRC - percentage of total L1 load misses that have reusable data on a remote L1.

S.No.	Suite	Benchmark	ABV.	Dataset	PerfX	μRC
1	MapReduce	Matrix Multiplication	mm	768 × 768 data points	9.86	4%
2	MapReduce	Similarity Score	ss	1024 × 256 data points	6.18	28%
3	Rodinia	Computational Fluid	cfd	200000 elements	6.17	51%
4	MapReduce	Page View Rank	pvr	21 MB	5.93	32%
5	Rodinia	Stream Cluster	sc	16384 points; 256 dimension	5.49	18%
6	Rodinia	Breadth-First Search	bfs'	1000000 nodes	5.18	3%
7	Rodinia	Wavelet Transform	dwt2d	1024 × 1024	4.96	7%
8	Parboil	Lattice-Boltzmann Method	lbm	120 × 120 × 150 data points	4.49	0%
9	MapReduce	K-Means	km	10000 × 3 data points; 24 clusters	3.85	24%
10	Rodinia	Hybrid Sort	sort	4194304 floating points	3.68	1%
11	Parboil	Breadth-First Search	bfs	8500000 nodes	3.57	6%
12	Rodinia	Particle Potential	lavaMD	7 × 7 × 7 boxes	2.81	1%
13	Parboil	2-D Histogram	histo	10000 × 4 dimension	2.63	1%
14	MapReduce	String Match	sm	4 MB	2.52	3%
15	Rodinia	Cardiac Myocyte	myocyte	100 instances	2.38	1%
16	Rodinia	Needleman-Wunsch	nw	2048 × 2048 data points	2.31	8%
17	Rodinia	Graph Traversal	b+tree	10000 nodes	2.21	25%
18	MapReduce	Inverted Index	ii	28 MB	2.19	2%
19	Rodinia	Particle Filter	pfloat	128 × 128 × 10	2.15	8%
20	Rodinia	Tracking Microscopy	leukocyte	176 MB	1.88	1%
21	MapReduce	Word Count	wc	96 KB	1.86	54%
22	Parboil	Sum of Absolute Diff.	sad	52 KB *vs*. 52 KB frame	1.76	3%
23	Rodinia	Speckle Reduction	sradv1	512 × 512 data points	1.74	15%
24	Rodinia	Speckle Reduction	sradv2	2048 × 2048 data points	1.70	16%
25	Parboil	Cartesian Gridding	mri-g	61 MB	1.49	2%
26	Rodinia	K-Means	kmeans	204800 data points; 34 features	1.47	0%
27	Rodinia	Matrix Decomposition	lud	2048 × 2048 data points	1.27	28%
28	Parboil	PDE Solver	stencil	512 × 512 × 64 input	1.23	6%
29	Rodinia	Heart Wall Tracking	heartwall	49 MB	1.19	0%
30	Rodinia	Back Propagation	backprop	65536 input nodes	1.10	3%
31	Rodinia	Thermal Modeling	hotspot	512 × 512 data points	1.07	29%
32	Parboil	Coulombic Potential	cutcp	96604 atoms	1.00	78%
33	Parboil	MRI Reconstruction	mri-q	64 × 64 × 64 data points	1.00	0%
34	Parboil	Angular Correlation	tpacf	10391 data points	1.00	19%

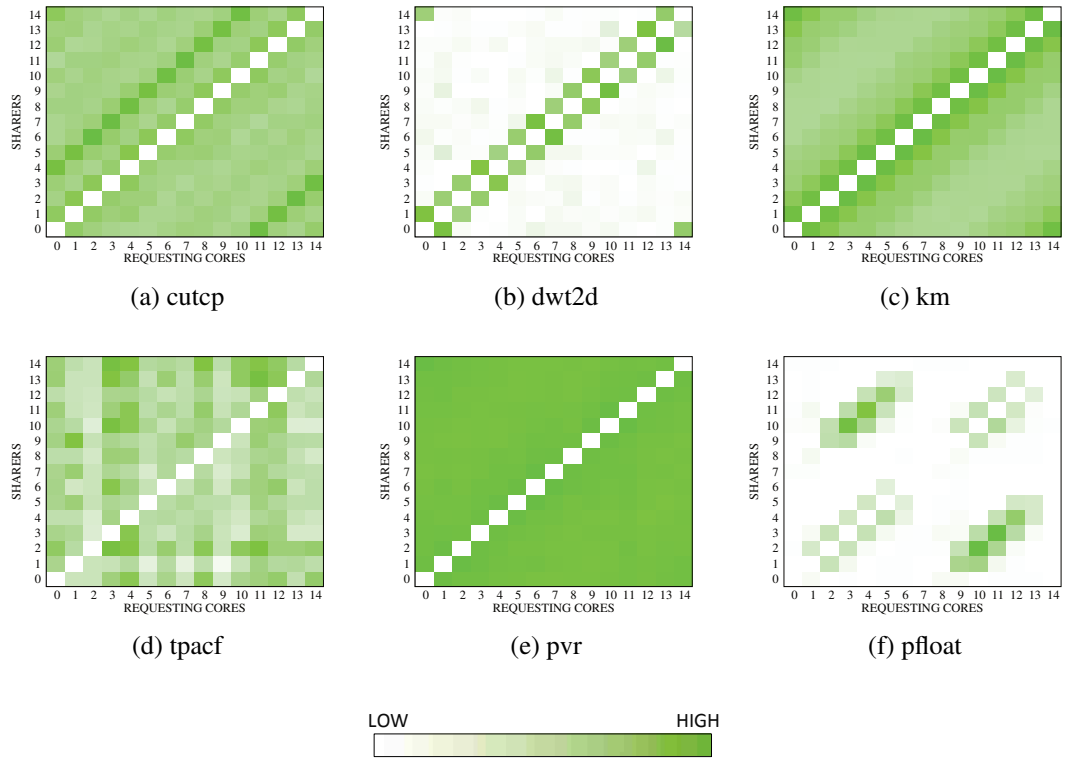
4.2.2 Workloads

For the purpose of this study, we use CUDA applications from three major general-purpose benchmark suites, *viz*., Rodinia (v3.0) [27], MapReduce [57] and Parboil [146]. We categorize the benchmarks according to their sensitivity to the memory hierarchy. Table 4.2 lists the benchmarks sorted by the speedup (PerfX) shown on a perfect memory system that has zero access latency to lower level memories and infinite bandwidth between memory hierarchies on a Fermi GPU.

A program is said to be memory-intensive if it comprises several instructions requiring long latency memory operations. As seen in the previous chapter, the performance of memory-intensive applications is usually bounded by the bandwidth to lower level memories. Therefore, applications with higher PerfX are considered as memory-bound or memory-sensitive applications as the magnitude of speedup on a perfect memory system essentially indicates the gravity of bandwidth problem in the benchmarks.

4.3 Need for Cooperation

Graphics and general-purpose workloads exhibit different memory access patterns. In traditional graphics applications, kernels typically operate on independent data of streaming nature. As a result, different thread blocks are executed in considerable isolation. On the other hand, general-purpose applications show varying amounts of reuse within the thread blocks and also at the boundaries with neighbouring thread blocks. For instance, in scientific application such as computation of Coulombic Potential (*cutcp*), atoms are organized in a 3D lattice. A sub-group of atoms constitute a thread block and the entire lattice is divided into multiple thread blocks. In order to compute the potential difference on the atoms at the edges and corners of a sub-lattice (or thread block), coulombic potential contributed by atoms from surrounding sub-lattices needs to be read, which requires sharing and reuse of data among neighbouring thread blocks. When such thread blocks are scheduled on different cores on a GPU, it results in inter-core reuse. In current GPUs, reuse across thread blocks on different cores can only be exploited by localizing the data on the L2 cache and not any closer. But in doing so, cores have to incur the congestion delays in L1-L2 interconnect, as well as the delays in the L2 access queues. Thus, for those applications that are bounded by the bandwidth to the lower level, it degrades overall performance by clogging the access path to the L2 cache.



(a) cutcp (b) dwt2d (c) km

(d) tpacf (e) pvr (f) pfloat

(g) Reuse Score

Figure 4.2: Heatmaps indicating inter-core reuse by cores on the x-axis for data cached on the cores on the y-axis. Dark spots in the heatmaps indicate high reuse between the corresponding cores at their x and y coordinates.

4.3.1 Inter-core Reuse

In order to quantify the degree of temporal and spatial reuse of global data between thread blocks, we analyse the L1 miss traffic of each core. In Table 4.2, we show the Reuse Coefficient (μRC), which is the percentage of miss requests received by the L2 cache from private L1 caches for addresses that reside remotely on at least one L1 cache. We see a maximum μRC of up to 78%, and an average of 14% across all benchmarks. High μRC for some benchmarks indicates that reuse requests from L1 caches form a large portion of traffic to L2. It is worth noting that we only consider it as reuse if the load miss address is cached on a remote L1 *at the time of the miss*.

In Figure 4.2 we further characterize the inter-core reuse patterns at the granularity of each core with every other core, providing deeper insight into the reuse dynamics. For brevity, we show the set of distinct observed patterns and omit those that replicate the patterns shown here. The x-axis indicates the cores that incur an L1 load miss and

the y-axis indicates the sharers for that miss. A dense area in the heat map at coordinate (x, y) indicates that a high proportion of load miss requests by core-x are cached by the L1 at core-y. For instance, *cutcp* shows a prominent reuse of data cached at a distance of 4 cores from the location of the miss; *dwt2d* shows a strong reuse between neighbours; *km* shows a gradual decline in reuse as we go further from the core; and *tpacf* shows considerable levels of reuse across all cores.

4.3.2 Efficacy of Cooperation

We have shown in the previous section that for general-purpose applications there is considerable reuse across L1 caches. We refer to those load requests as *reuse requests* that miss in the local L1 but hit in a remote L1. By removing such reuse requests (also quantified as μRC) from the pool of total misses going to the L2 cache, we can reduce the pressure on L2 bandwidth. In order to assess the efficacy of reducing the bandwidth demand on the overall performance, we begin by examining the performance improvement when reuse requests do not congest the access path to L2. In these cases, reuse requests are instead serviced cooperatively within the L1s with varying remote L1 access latencies, or *reuse latencies*. Since applications with low μRC are not expected to show any change, we focus on benchmarks with high μRC. Later, we demonstrate the effect of our final proposal on applications with low or zero μRC as well.

Figure 4.3 shows the speedup due to cooperation, and demonstrates a noticeable improvement in performance, specifically for memory-intensive applications with high μRC. For instance, *cfd* and *pvr* show performance improvements of up to 73% and 38% respectively. Both of these applications are severely bounded by the memory bandwidth and at the same time exhibit high reuse. On the other hand, despite high reuse in *cutcp* and *hotspot*, there is no significant gain in IPC since bandwidth is not critical for these benchmarks.

Another *key observation* in this study pertains to the variation of performance as a function of remote L1 access latency. We observe that the performance improvement in the region between 0-80 cycles is fairly stable, with the average IPC gain only changing from 21.5% to 18.8%. This is because in this region, latencies to remote L1s can be effectively hidden by multithreading on the cores. Moreover, reduced congestion in the L2 access path and faster responses to reuse requests (compared to L2 accesses) improves the average number of active warps on the cores. This boosts the ability of the cores to further mask the memory access latencies. Due to these effects, reuse

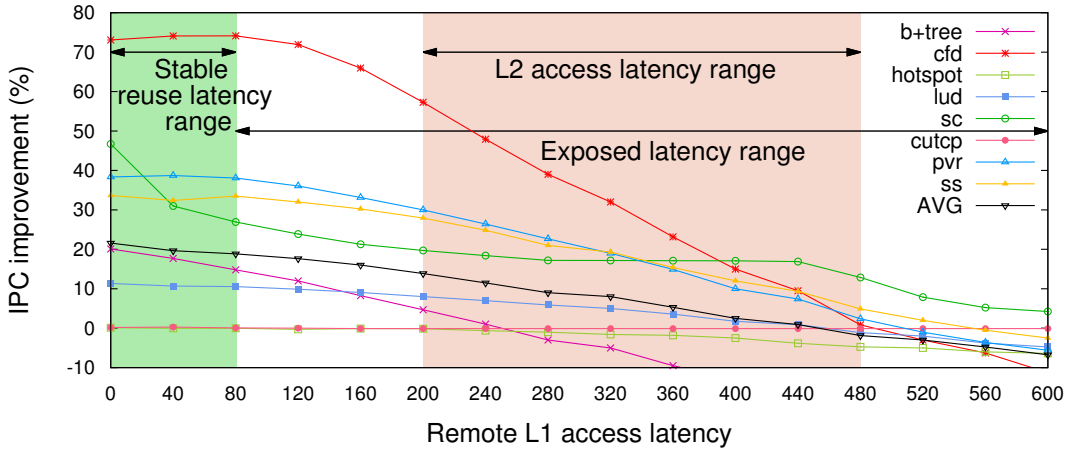


Figure 4.3: Speedup of cooperation with varying remote L1 access latencies.

latencies up to 80 cycles are effectively hidden by multithreading and do not determine the execution time. However, on further increasing the reuse latencies, performance improvement starts to degrade more rapidly. In fact, the IPC gain returns to nearly 0% when the reuse latencies are varied in the range of L2 access latencies (around 300 cycles). This is because latencies for reuse requests get increasingly exposed and can no longer be hidden by multithreading, despite reduced congestion.

In summary, these initial results indicate that for memory-bound applications, when there is considerable reuse of data across L1 caches, cooperation among the private L1 caches can result in a considerable speedup (up to 21.5% on average). Notably, the observed performance improvement is fairly stable in the reuse latency range of 0-80 cycles.

4.4 Cooperative Caching

In the previous sections, we observed a potential for cooperative caching on GPUs and assessed its efficacy. We now propose a cooperative caching framework to use the private L1 data caches in an aggregate manner. We begin by formalizing the above discussion and analysing the parameters that contribute to the L2 access latencies for L1 miss requests. Later, we propose a cooperative caching scheme and discuss the architectural details.

4.4.1 Analytical Model

Here we present a simple analytical model to explain the conditions under which reuse delivers a performance gain. Firstly, in the absence of cooperation between L1s, let l_O be the average memory latency to access the shared L2 cache. Secondly, with cooperation between L1s, let h_{reuse} be the fraction of L1 misses that hit in a remote L1 cache. Furthermore, let l_{reuse} be the average hit latency for accesses to remote L1s. As a consequence of reduced congestion in the L2 access path due to remote L1 hits, let δ_{cong} be the reduction in L2 access latency. And finally, let $\delta_{overhead}$ be the cooperation overhead borne by those requests that do not have a shared copy. Therefore, the new average memory latency to L2 upon enabling cooperation, l_C, can be obtained via Equation 4.1.

$$l_C = (l_O - \delta_{cong} + \delta_{overhead})\ (1 - h_{reuse}) + l_{reuse}\ h_{reuse} \tag{4.1}$$

$$\left.\begin{array}{l} l_{reuse} < l_O \\ \delta_{overhead} < \delta_{cong} \end{array}\right\} \text{Criteria for useful cooperation} \tag{4.2}$$

In order to derive gain from cooperative caching, l_C must be minimized. Therefore, remote L1 accesses for reuse requests must take less time than a normal L2 access, *i.e.*, $l_{reuse} < l_O$. Additionally, we have already seen in Figure 4.3 that the maximum gain from cooperation is sustained in the lower reuse latency range, *i.e.*, $l_{reuse} \in (0, 80)$. Finally, for the remaining L2 accesses, the cooperation overhead must be less than the benefit obtained from reducing the congestion in the L2 access path, *i.e.*, $\delta_{overhead} < \delta_{cong}$. A combination of above conditions will result in a lower average L2 access latency, *i.e.*, $l_C < l_O$.

How should we go about implementing the cooperative caching framework? Following the approach of traditional multicores, a central directory in the L2 cache [91, 2, 74] can be used to store information about the sharers. However, maintaining a directory *as part of the L2* will not mitigate the existing bandwidth problem in *accessing the L2*, and instead, will only worsen it. This is because the additional control and update traffic to the central directory will further increase the bandwidth demand to the L2 cache. Alternatively, an approach along the lines of cooperative caching schemes for CPUs [23, 24, 59] may be used. Such schemes aim to minimize hop latencies to find a sharer and retrieve data using a highly interconnected network of L1 caches. However, since we have demonstrated that we have a considerable leeway of around 80 cycles to fetch the shared data from a remote L1, such an aggressive scheme to find a sharer is an overkill for GPUs.

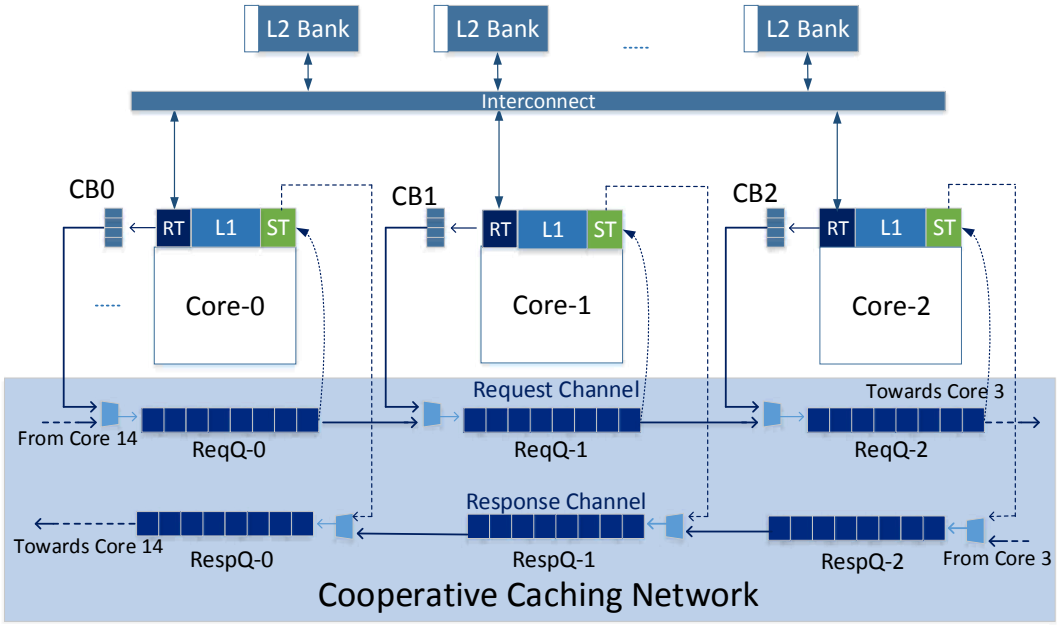


Figure 4.4: Cooperative Caching Network.

In view of the above discussion, we propose a lightweight ring-based Cooperative Caching Network. A ring topology is the lowest degree network and requires the fewest number of inter-core connections. It is also lowest in terms of logical complexity and power consumption as all core-to-core connections will be near-neighbour, and therefore, the wires will be short. In addition, all routers in a ring are simple multiplexers, which are more energy efficient than complex crossbar routers. As we have shown that GPUs can tolerate reuse latencies gracefully up to 80 cycles, a ring topology appears to be a cost-effective solution, as it allows us to trade-off higher latencies for simplicity and short wires, *i.e.,* lower power consumption and die-area cost.

4.4.2 Architecture

In our proposed scheme, we facilitate the communication between neighbours by connecting the private L1 caches in a ring via our Cooperative Caching Network (CCN). The CCN comprises two different channels, *viz.*, request channel and response channel. The request channel comprises a network of Request Queues or ReqQ while the response channel comprises a network of Response Queues or RespQ. As shown in Figure 4.4, each L1 has an independent pair of the aforementioned queues to allow the cache to participate in cooperative caching. The L1 caches interact with their home queues via CCN Buffers (CB), which hold the tag and Core-ID for the load misses, until the

CCN is ready to accept a request. A new miss request, after allocating a cache line in the L1 cache as done in the baseline architecture, enters the local Request Queue. Thereafter, it travels around the request channel by hopping on other Request Queues and probing the different L1 caches on its way. If a remote copy is found in one of the nodes, the response from the hit node is sent back to the requesting core in a similar way by hopping in the reverse direction via the Response Queues at each core. Finally, after reaching the requesting core, the previously allocated cache line is filled with the response data and the miss is serviced. Note that a remote L1 copy is considered for sharing only if it is not pending on a cache-fill for the requested data at the time of lookup. In other words, hits-under-miss on outstanding miss requests are not considered for sharing in the CCN.

More specifically, upon incurring an L1 miss for global data and subsequently allocating a cache line in the L1 cache, each core pushes the miss tag information into its CB along with the Core-ID, where the request waits until the corresponding ReqQ is ready to accept a new request. At every cycle, valid entries at the head of the ReqQ lookup the corresponding L1 cache (if it is not the home core of that request) before hopping on to the next ReqQ. If the request travels back to the requesting core without a reuse copy, it is finally sent to L2. However, if a sharer is found, the sharing core enqueues the response to its RespQ. The response travels back to the requesting core and fills the allocated cache line in the L1 cache, thereby avoiding an L2 access. If request queues get full due to congestion, the CB eventually stops accepting new miss requests. In such a scenario, the L1 load misses go *directly to L2* until the CCN can start accepting new requests again.

Prioritization Policy for Queues: Each queue in the CCN has a corresponding input multiplexer to select one of the entries out of the two possible input sources. In the request channel, a ReqQ can either accept a new miss request from the home core via CB, or a forwarded request from a preceding ReqQ. In our proposal, we prioritize an older request (from ReqQ) over a new one (from CB). This helps in preventing over-subscription of CCN to new L1 misses by allowing the previously accepted requests to pass-through. Therefore, it minimizes the roundtrip overhead ($\delta_{overhead}$) in CCN for subscribed requests. Repeated unsuccessful attempts to inject a new request in the CCN due to the above prioritization, thus causes the CB to get full and hence, deflects the L1 misses *directly to L2*, allowing the CCN to recover from congestion.

In response queues, however, we prioritize a new cache response (from Core) over an older response (from RespQ). This is because response queue latencies do not contribute

to $\delta_{overhead}$ but contribute to the reuse latencies l_{reuse}, which has comparatively more relaxed requirements (shown in Figure 4.3). More importantly, if the response of a new remote hit is not accepted by the response queue, the tag entry at the head of the corresponding ReqQ that caused the hit is not popped, potentially stalling the entire request network and increasing the $\delta_{overhead}$ in the request channel.

CCN Memory Consistency: The CCN mechanism conforms to the existing memory consistency model supported by Fermi. CUDA provides two types of load instructions [128] — a normal load cached at L1 (*ld.ca*) and a direct load to L2 bypassing the L1 (*ld.cg*). Due to the *write-through, no-write-allocate* policy of the L1 cache, a write causes the matching cache line in L1 to be invalidated, thereby causing the most recent value to reside in L2. However, due to a weak memory model [40, 4] and absence of coherence in GPUs, an *ld.ca* accessing L1 on a different core can return a stale value. Litmus tests [4] have shown that due to weak consistency, an *ld.ca* load may return a stale value on the same core as well, even if preceded by an *ld.cg* to the same address (CoRR). CCN adopts similar weak memory ordering semantics for *ld.ca* loads; indeed, an L1 miss can return a stale value by snooping other cores via CCN, instead of reading the L2 which may have the latest value. However, since a baseline GPU guarantees reading the most recent value for *ld.cg* loads, *CCN does not intercept such loads, and hence, does not further weaken the memory model.* In other words, when the programmer or the compiler uses *ld.cg* loads to bypass the L1, the current memory model ensures the most recently written value is returned — a correctness guarantee also provided by CCN.

4.4.3 Shadow Tags

Since each L1 now services additional tag lookups for CCN requests, such remote lookups could affect the performance of local cache accesses. To eliminate the interference of remote lookups on local requests, we duplicate the tags of the L1 data cache in a separate set of Shadow Tags (ST) adjacent to each L1. The shadow tags always contain an identical copy of the L1 tags, which is achieved by always writing tag updates to both sets of tags simultaneously. As a result, concurrent reads at independent addresses can then take place to L1 tags and shadow tags, from the local core and remote lookups respectively. Therefore, the shadow tags decouple the performance of each local cache from interference of CCN traffic. However, if a shadow tag lookup succeeds, then the remote access makes a regular L1 access to retrieve the data it needs. This steals a cycle

from the L1 data cache, which is taken into account in our performance model.

Overhead: For the largest L1 data cache configuration of 48 KB with 128 byte line size, we require 24 upper address bits per tag, assuming 40-bit physical addresses [115], plus one valid bit. As the L1 data cache is 4-way set-associative, the shadow tags are arranged as 96 sets of four 25-bit tags in 96×100 single-ported tag memory.

Way 0		Way 1		Way 2		Way 3	
V_0	Tag_0[39:16]	V_1	Tag_1[39:16]	V_2	Tag_2[39:16]	V_3	Tag_3[39:16]

Therefore, the net storage overhead of the shadow tags is 1200 bytes per SM, amounting to a total of 17.5 KB for a 15 core GPU that we consider in our study. Although, each remote access has to be checked in multiple shadow tags, these shadow tag memories are small and can be constructed from low-leakage high-density bit-cells without impacting the overall cycle time of the ring interconnect.

4.4.4 Request Throttler

In order to prevent those cores that do not exhibit any inter-core reuse from congesting the CCN, we introduce a Request Throttler (RT) at each core. The purpose of RT is to throttle the requests from entering the CCN when prior routing of misses to CCN proves to be below a threshold level of effectiveness. In such scenarios, RT diverts the remote lookup requests directly to L2 cache. In order to do this, each RT periodically samples the CCN performance parameters and at the end of the sampling period, computes the success rate in routing its load misses to CCN during the sampling interval. The success rate is determined by the ratio of hits in the CCN to the total number of requests injected in the CCN by the corresponding L1 cache. If the success rate is below the threshold, the L1 cache bypasses the CCN until the next sampling interval, and performs the load miss by sending the request directly to L2 cache. However, the shadow tags of the throttled cores still participate in the lookup for other requests in the CCN.

To illustrate the working of RT further, we define the sampling interval as t_S and the periodicity of sampling as t_P where $t_S << t_P$. Therefore, the entire period of execution is logically divided into multiple epochs of duration t_P. We also define H_{min} as the minimum hit rate required in the CCN in order to derive a benefit from cooperation. At the beginning of an epoch of interval t_P, each core begins by routing the load misses to CCN for a fixed sampling duration of t_S. During the t_S interval, RT collects the

statistics about the number of requests injected in the CCN (N_{total}) and the number of hits observed for its requests (N_{hits}). At the end of the sampling duration, RT computes the hit rate (h_{reuse}) in the CCN, *i.e.,* $h_{reuse} = N_{hits}/N_{total}$. If $h_{reuse} > H_{min}$, RT continues to inject requests in the CCN for the remaining duration of (t_P - t_S) in the current epoch. On the other hand, if $h_{reuse} < H_{min}$, RT disables the routing of requests to CCN for the remaining duration of the epoch. After the current epoch ends, N_{hits} and N_{total} are reset and RT repeats the entire process again for the new epoch. Therefore, with the help of RT, we improve the average success rate of sending a load miss to CCN by preventing those cores from cooperating that are not working on potentially reusable data, during specific epochs of execution.

4.4.5 Working Example

In this section, we further illustrate the working of CCN. Figure 4.5 shows the flow of requests within the CCN. In this example, Core-0 incurs a load miss for a global data in its private L1 cache and allocates a cache line in the L1 cache. In the baseline architecture, this L1 miss would be directly routed to the L2 cache. However, with our scheme, the miss request can either go to the CCN or to the L2 cache. RT takes this decision for that particular core on the basis of the statistics collected over the most recent sampling interval, t_S. In this example, we assume that h_{reuse} for Core-0 and Core-1 suggests healthy reuse ($> H_{min}$), and therefore these cores continue to use CCN. However, Core-2 observes a low reuse in the recent t_S interval, thereby routing all requests directly to the L2 cache in the current epoch.

In order to service a miss at Core-0 via the CCN, the tag and Core-ID of the load request are pushed ❶ onto the corresponding CCN Buffer, CB0. Based on the input prioritization policy for ReqQ, the new tag waits in CB0 until it acquires priority and is accepted ❷ by the ReqQ-0. Upon reaching the head of ReqQ-0, the miss request does not perform a lookup in the ST of Core-0 as it is the home core of the miss request, and therefore it is directly passed to ReqQ-1 ❸. Upon reaching the head of ReqQ-1, it performs a lookup ❹ in ST of Core-1. Assuming it is a hit in Core-1, the ST receives the cache line from the corresponding L1 cache and enqueues the response ❺ in RespQ-1, if the RespQ-1 is not full. On the other hand, if the RespQ-1 is full, the response is stalled, thereby preventing the tag at the head of the ReqQ-1 from getting popped. Once the response reaches the head of the queue at RespQ-1 and acquires priority to enter the next queue, it is pushed into RespQ-0 ❻. Since Core-0 is the home

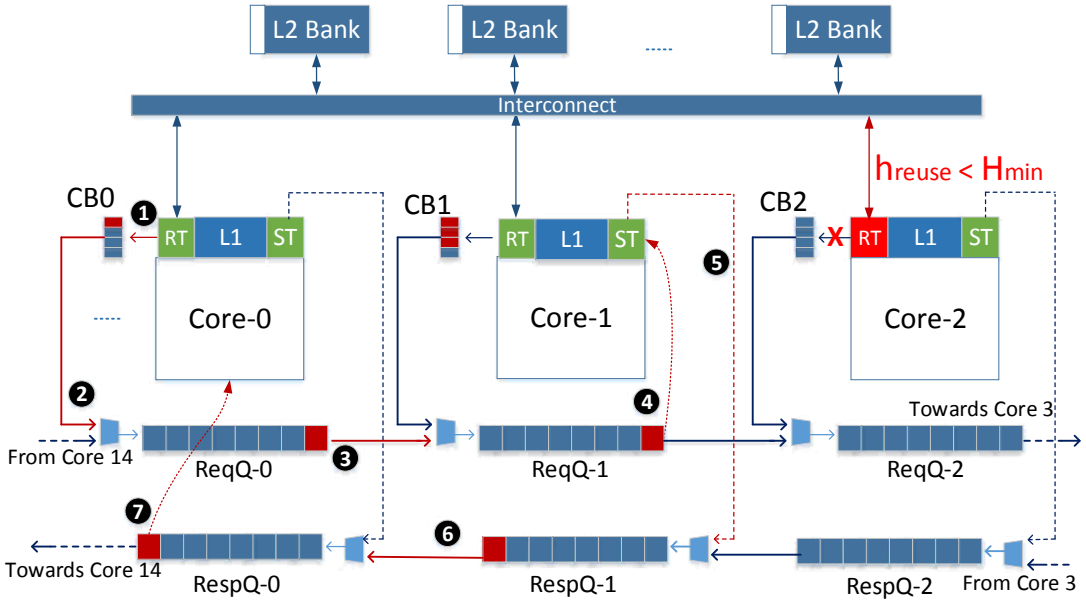


Figure 4.5: Working of the Cooperative Caching Network with Request Throttling.

core of the response, the new entry to RespQ-0 is bypassed to the head of the queue and the response is serviced ❼ by filling the allocated cache line in the L1 cache of Core-0, hence completing the request-response cycle.

4.5 Evaluation

In this section, we discuss the implementation of our proposed architecture and demonstrate the results.

4.5.1 Implementation

For the purpose of this study, we implement and evaluate two flavours of our proposed architecture, *i.e.,* CCN-B and CCN-RT. CCN-B is our baseline CCN architecture which includes a pair of queues and shadow tags at every node of the network. Whereas in CCN-RT, we add the request throttling feature to the baseline CCN architecture. Table 4.3 summarizes the design parameters for CCN-B and CCN-RT.

In our implementation, we choose the sampling interval and the periodicity of sampling as 1 million and 10 million instructions respectively. This is based on the observation that most benchmarks show a single-phase sharing across the entire application. Hence, it allows us to sample for a short duration to get a fairly accurate hint

Table 4.3: CCN parameters

Parameter	Value
CCN Buffer	8-entry, 30 bits per entry (26 bits Tag + 4 bits Core-ID)
Request Queue	8-entry, 30 bits per entry
Response Queue	8-entry, ∼128 bytes per entry (cache line + Core-ID)
CCN ring	4-byte request channel; 32-byte response channel; 1.4 GHz
Shadow Tag	1200 bytes size (modelled upon 48 KB L1 data cache)
t_S	1 million instructions
t_P	10 million instructions
H_{min}	0.05 (5 percent hits)

for a large duration that follows the sampling interval. Furthermore, on the basis of our sensitivity studies, we select the threshold hit rate (H_{min}) as 5%, *i.e.,* the minimum number of hits required to derive benefit from cooperative caching. We also observe in our experiments that small 8-entry Request and Response Queues provide optimal results. Lastly, the request and response channels in CCN are configured to flow in opposite direction. This is because our experiments show that in such a case, servicing reuse requests takes an average of 10 hops compared to a fixed 15 hops when both the channels propagate in the same direction.

4.5.2 Experimental Setup

We model the Cooperative Caching Network on GPGPU-Sim (version 3.2.2) [10] to simulate a Fermi-like GPU with the configuration parameters listed in Table 4.1. For energy and area simulations, we use GPUWattch [98], a McPAT based power model integrated in GPGPU-Sim. All CCN transactions have been modelled at *cycle-by-cycle accuracy* in the simulator which includes queuing delays in the request and response channels, CCN congestion and L1 cycle stealing by shadow tag accesses. We run all the benchmarks to completion, or until they execute 16 billion instructions, whichever comes first.

4.5.3 Results

We begin by evaluating the overall performance improvement with our proposed schemes for benchmarks that exhibit inter-core reuse (μRC > 10). We also show

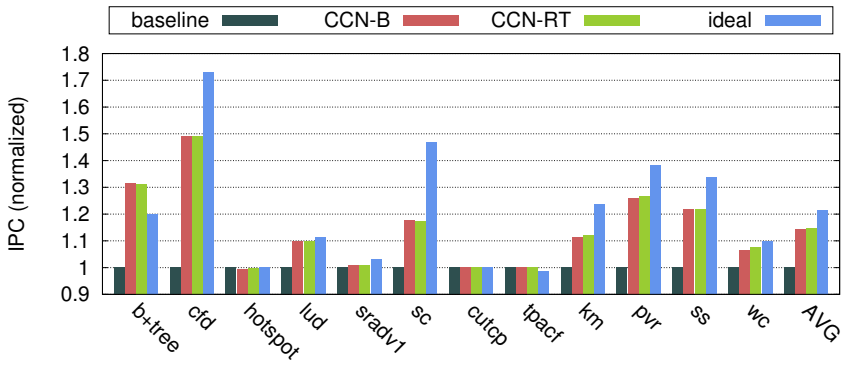


Figure 4.6: Speedup for applications with μRC $>$ 10

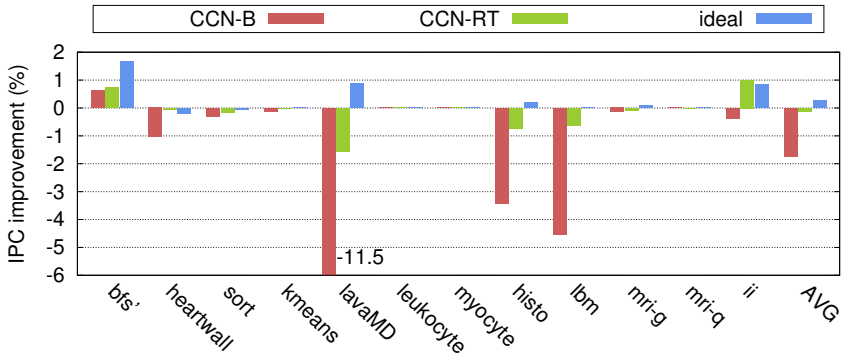


Figure 4.7: Percentage improvement in IPC for applications with μRC $<$ 3

the neutrality of our scheme for benchmarks with little or no reuse (μRC $<$ 3). Later we assess the finer parameters for the former set of benchmarks, as applications with inter-core reuse are the primary motivation for this study. We do not show the benchmarks between this range, as results of the above categories are good indicators of the trend in the rest of the benchmarks. We also compare the results of our proposed schemes, *i.e.*, CCN-B and CCN-RT, against an *ideal* cooperative caching configuration that services all of the remote hits with zero latency, without incurring any overheads of cooperative caching.

Performance: In Figure 4.6, we show the speedup with CCN-B and CCN-RT for applications that exhibit reuse. Over the baseline configuration, we observe an average improvement of 14.5% with CCN-B and 14.7% with CCN-RT. Memory-bound applications such as *cfd*, *ss* and *pvr* show higher speedup compared to non-memory-bound applications as they are more sensitive to bandwidth bottlenecks. *b+tree* shows a higher improvement than ideal case due to the timing variations in scheduling warps. Such an aberration is also caused by a higher number of hits-under-miss on cache lines

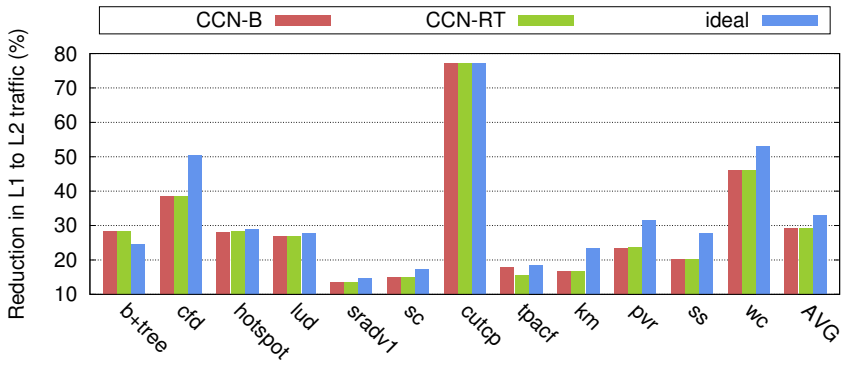


Figure 4.8: Percentage reduction in L1 to L2 traffic

allocated for on-going remote L1 accesses, which does not occur in the ideal scenario due to zero cycle latency for remote L1 accesses.

We also assess the impact of cooperative caching on applications that show little or no reuse. For such applications, cooperative caching adds an extra roundtrip overhead of going through the CCN. This is because due to low μRC, most requests end up going to L2 cache after an unsuccessful traversal in the CCN. In such cases, Request Throttler helps in preventing the L1 misses from incurring the CCN overhead when there is little or no reuse. In Figure 4.7, we show that with CCN-B, we see a degradation of up to 11.5% and an average degradation of 1.7% compared to the baseline GPU. However, with CCN-RT, the maximum degradation reduces to 1.5% with an overall average of 0.1%.

L2 Cache Bandwidth Demand: In Figure 4.8, we demonstrate the effectiveness of our proposed technique in mitigating the L2 cache bandwidth bottleneck. On average, CCN-RT reduces the traffic to L2 cache by 29% compared to the baseline GPU. It is in close proximity to the ideal-case average of 33% indicating that most of the reuse hits on remote L1 caches are captured by the proposed architecture. The minimal difference between CCN-B and CCN-RT demonstrates that, while throttling diverts most of the non-reuse traffic directly to L2 cache, it does not reduce the number of potential hits in the CCN. If it would divert useful reuse requests to L2 cache, bypassing the CCN, then we would see a lesser reduction in L2 traffic with CCN-RT compared to CCN-B.

Average Memory Latency (AML): In Figure 4.9, we see an average reduction of 24% in AML with our proposed CCN-RT architecture for applications that show reuse. We observe that *cutcp* shows the maximum reduction of 65% in AML due to a high μRC of 78%. However, it does not translate into performance gain due to its non-memory-bound nature.

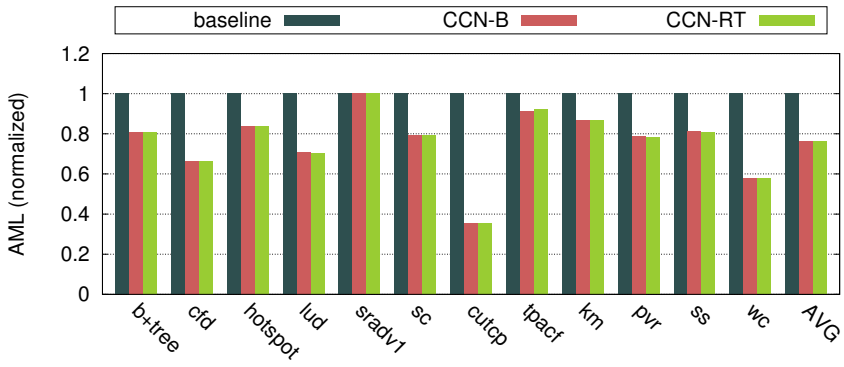


Figure 4.9: Normalized average memory latency

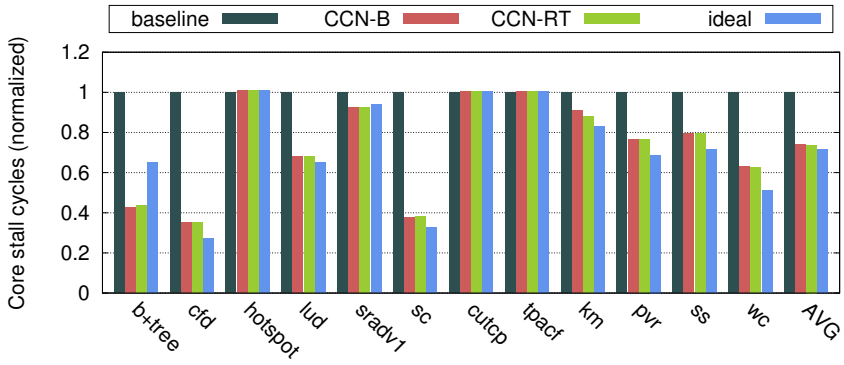


Figure 4.10: Normalized core stall cycles

Core Stall Cycles: We observed in the above results that performance is gained by mitigating the bandwidth problem (indicated by L2 traffic), and by servicing the misses in less time (indicated by AML). This is because cores now spend less time waiting for memory. Therefore, we assess the impact of our proposal on the total number of cycles for which the cores are stalled. In Figure 4.10, we observe a significant reduction in core stall cycles for memory-bound applications such as *cfd* and *sc*, while no degradation is seen for non-memory-bound applications like *cutcp* and *tpacf*. On average, we reduce the core stall cycles by 26%, which is in close proximity to the ideal reduction of 28%.

Off-chip Memory Traffic: In order to dissociate the effects of L2 and off-chip bandwidths on the overall performance gain, we analyse the change in off-chip memory traffic. As shown in Figure 4.11, we see that for most applications there is *no visible difference* in the traffic to off-chip memory, indicating that the entire performance improvement can be attributed to the mitigation of bandwidth bottleneck between private L1s and the shared L2. Therefore, it can be inferred for most benchmarks that in the baseline architecture without CCN, the reuse requests mostly hit in the L2 cache,

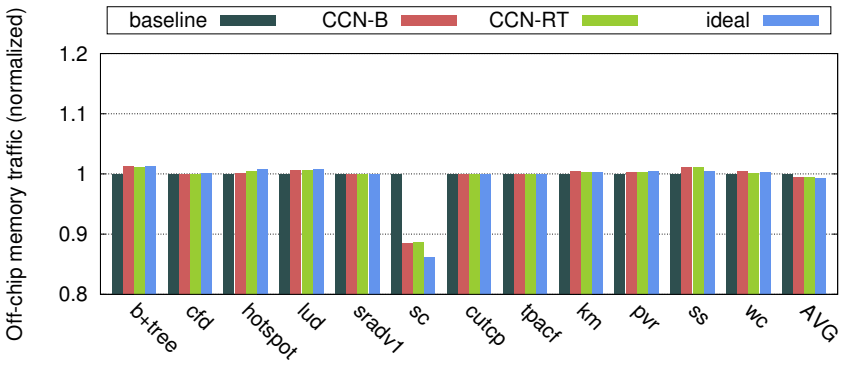


Figure 4.11: Normalized off-chip memory traffic

thereby burdening the L2 cache bandwidth with duplicate requests. However, in *sc* we notice a reduction in DRAM traffic by 12% with CCN-RT. This indicates that for *sc* a significant portion of reuse requests to L2 also misses in the L2 cache, adding to the DRAM traffic. As a result, upon removing the reuse requests to L2 cache with the help of CCN in *sc*, not only the traffic to L2 cache is reduced, but also the traffic to DRAM is reduced. Therefore, the performance benefit in *sc* with CCN-RT can be attributed not only to the mitigation of L2 bandwidth bottleneck, but also to the mitigation of DRAM bandwidth bottleneck.

Summary: In the above results, we observed that for applications which exhibit reuse, we are able to reduce the traffic to L2 cache by 29% while also reducing the average memory latency by 24%. As a consequence of the above improvements, we reduce the average core stall cycles by 26%, which translates into an average performance improvement of 14.7%.

4.5.4 Hardware Cost

Area: We use GPUWattch [98] to estimate the area of our proposed architecture. We use the existing components in GPUWattch to model the CCN components, after appropriate scaling wherever necessary. CCN adds an area overhead of 4.38 mm^2 for the ring interconnect and the shadow tags (corresponding to the largest L1 data cache configuration) at 40 nm technology. Other storage units such as CCN Buffers and Request/Response Queues add another 4.82 mm^2. This amounts to an overall increase in die area by 1.3% compared to the baseline processor area of 700 mm^2.

Energy: With CCN, cores are stalled for fewer cycles, thereby *reducing the leakage power*. In addition, fewer packets require routing at the energy-inefficient crossbar

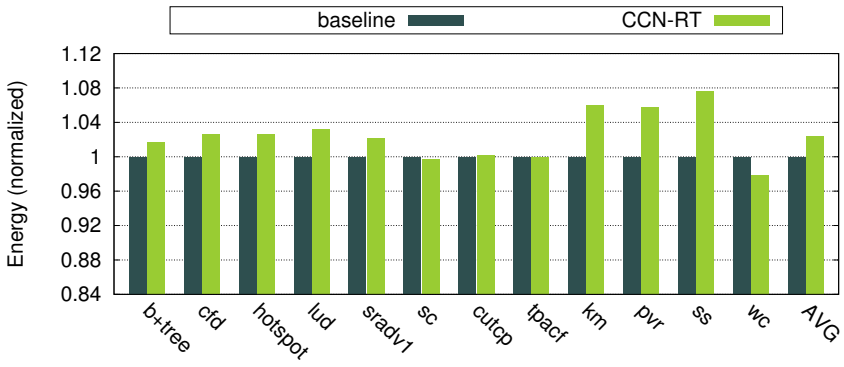


Figure 4.12: Energy dissipation with CCN

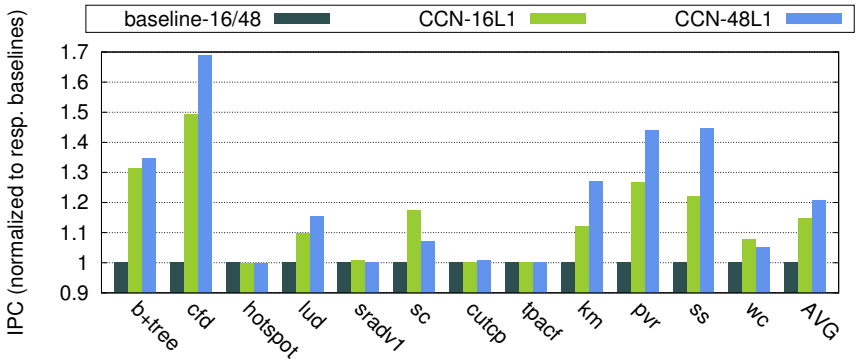


Figure 4.13: Speedup with varying L1 cache size

routers. Also, lower traffic to L2 leads to lower energy consumption by the NoC. However, high shadow tag lookups for remote cache accesses normalizes other energy gains of the CCN, resulting in an average energy overhead of 2.5% (Figure 4.12).

4.5.5 Sensitivity Analysis

L1 Cache Size: As Fermi offers configurable L1 cache sizes of 16 KB and 48 KB, we analyse the sensitivity of our proposal to L1 cache size. As shown in Figure 4.13, with an L1 cache size to 48 KB we observe an average IPC gain of 20.6% with CCN, compared to 14.7% with CCN with 16 KB L1 (over their respective baselines with CCN). This is due to the following reason: although increasing the L1 cache size reduces the number of capacity/conflict misses, thereby reducing the opportunities to find remote L1 hits in the CCN, we observe that a larger L1 significantly increases the likelihood of finding a remote L1 sharer for a *compulsory miss*. Therefore, due to significant increase in utility of CCN for compulsory misses on increasing the L1 size (which dominates the decrease in utility of CCN due to lower conflict/capacity misses), we observe a higher improvement in performance with larger L1s.

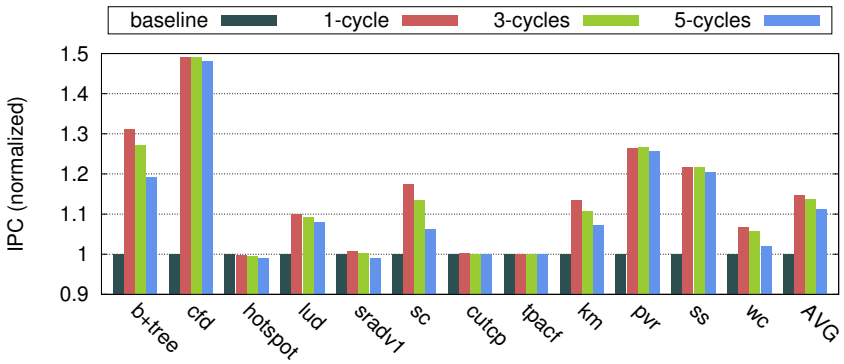


Figure 4.14: Speedup with link latencies of 1, 3 and 5 cycles

Link Latency and Frequency: In this study, we analyse the performance impact of interconnect latencies for every hop on the CCN ring. This is done by varying the core-to-core transfer latency from 1-5 cycles (*i.e.*, 15-75 cycles to traverse the entire ring). For a 700 mm^2 chip, each hop is approximately 3.5 mm of on-chip distance, and therefore 1-5 cycles at 1.4 GHz is a reasonable window to complete the transfer [15]. It is worth noting that varying the CCN link latency also captures the effect of running the CCN ring at a fraction of core frequency. Therefore, this study shows the performance variation on using the CCN ring at up to $1/5^{th}$ the core frequency (280 MHz).

In Figure 4.14, we see that for most applications, the IPC gain is fairly resilient to increasing link latencies (or decreasing ring frequencies). For instance, *cfd* shows a marginal reduction of 1% when the latency increases from 1 to 5 cycles. Only a minority of applications show visible reductions in the gain as link latency increases. For example, the IPC gain of *b+tree* drops from 31% to 19%, although it still maintains a modest overall improvement in performance. On average, we see IPC improvements drop from 14.7% to 13.6% as latency is increased from 1 to 3 cycles, settling further at 11.2% when the link latency is increased to 5 cycles. These results indicate that our proposed scheme is fairly robust to increasing latencies in the ring interconnect (as well as increasing distance between the neighbouring cores).

SIMD Lane Width: Each core in NVIDIA's Fermi GPU consists of a 32-lane SIMD unit, each lane capable of executing one floating-point or arithmetic instruction per clock. In this study, we analyse the impact on CCN of increasing the SIMD lane width. In Figure 4.15, we plot the performance gain with CCN-RT on baseline configuration with varying SIMD lane width of 32 (ccn-32), 64 (ccn-64), 128 (ccn-128) and 192 (ccn-192), each normalized to their respective baselines. On average, the performance gain drops modestly from 14.7% to 13.6% on increasing the SIMD lane width from 32 to 64, settling further at 11.4% and 10.2% with SIMD lane widths of 128 and

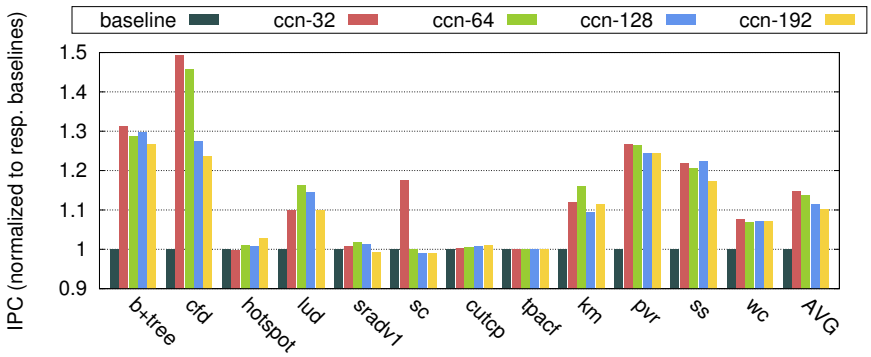


Figure 4.15: Speedup with varying SIMD lanes

192 respectively. Although the minor reduction in CCN gain is due to the increased latency tolerance provided by additional SIMD lanes, cooperative caching continues to provide considerable benefits for memory-intensive applications. This is due to the fact that by increasing the number of SIMD lanes or the compute capability of the cores, only compute-bound applications are expected to show significant speedups and a higher overlap of memory latencies with computation. In contrast, memory-intensive applications lack independent instructions and continue to be constrained by memory resources. Therefore, additional compute resources for memory-intensive applications provides only limited additional latency tolerance to the cores due to which cooperative caching continues to be useful in reducing memory latencies that lie in the critical path. However, some benchmarks such as *lud* and *km* also show momentary improvement in performance gain with CCN on increasing the SIMD lane width. We observe that this is because with wider SIMD lanes, higher number of threads perform memory instructions on each cycle, issuing higher number of requests that may exhibit reuse, thereby amplifying the utility of CCN in reducing the traffic that could lead to even higher congestion.

4.5.6 Discussion

In future, scalability of the CCN can be addressed by a hierarchical implementation of the proposed ring network [61, 129]. A sub-CCN-ring that contains the requesting core can inquire other sub-CCN-rings in *parallel*, thereby decomposing the serial latency of traversing the high number of cores into concurrent transactions to multiple rings. In addition, as coherent caches are imminent with future architectures [109, 127, 143], CCN can also act as a substrate for implementing cache coherence by providing a means for inter-core communication.

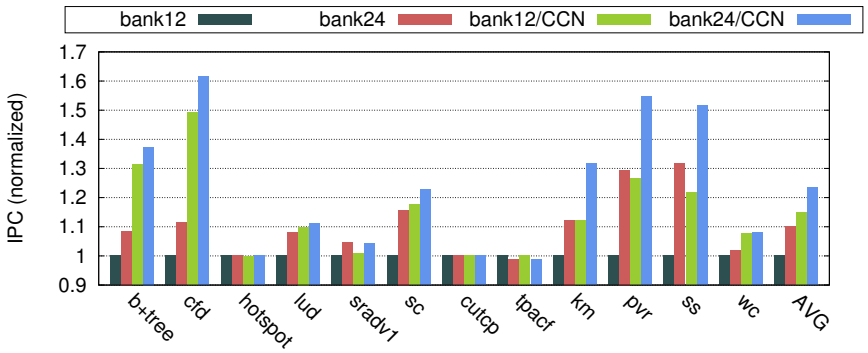


Figure 4.16: Speedup with 2× L2 banks and CCN

4.6 Comparative Study

In this section, we perform a quantitative and qualitative comparison of CCN with alternative techniques that address the bandwidth bottleneck in GPUs.

4.6.1 Increasing L2 Banks

An alternative technique to increase the bandwidth to L2 is to increase the number L2 banks. However, increasing the banks only reduces the congestion in the access path to L2 whereas CCN, in addition to reducing pressure on L2 bandwidth, provides a significantly faster response for a fraction of miss requests. In our experiments, we observe that CCN services the reuse requests in 42 cycles (l_{reuse}) on average, for 29% misses (h_{reuse}) that hit in CCN. For the remaining L2 accesses, CCN adds a roundtrip overhead of 54 cycles ($\delta_{overhead}$). It also reduces the congestion overhead to L2 by 78 cycles (δ_{cong}). Considering that the average L2 access latency without the CCN is 300 cycles (l_O) and substituting the above values in Equation 4.1, the average L2 access latency with CCN is computed to be 208 cycles (Equation 4.3).

$$l_{C(CCN)} = (300 - 78 + 54) \times 0.71 + (42) \times 0.29 = 208 \quad (4.3)$$

$$l_{C(2\times)} = (300 - 80 + 0) \times 1.0 = 220 \quad (4.4)$$

$$l_{C(CCN/2\times)} = (300 - 117 + 54) \times 0.71 + (42) \times 0.29 = 180 \quad (4.5)$$

However, increasing the L2 banks only reduces δ_{cong} (though marginally more than CCN for some benchmarks), but requires all accesses to go through the L2 access latency, albeit via reduced congestion. Upon substituting corresponding values in Equation 4.1, the reduced L2 access latency is computed to be 220 cycles (Equation 4.4). As a result,

in Figure 4.16 we observe an average performance improvement of 10.2% upon a 2× increase in L2 banks from 12 to 24. In contrast, CCN implemented with 12-bank L2 configuration shows a higher improvement of 14.7% (with *cfd* performing 34% better with CCN than with 2× L2 banks).

Importantly, CCN is *partly orthogonal* to increasing the banks at L2. This is because, in addition to reducing the δ_{cong} further, CCN adds the benefit of faster access to reuse requests. The average L2 access latency in Equation 4.1 for a CCN architecture on a 24 L2 banks configuration is computed to be 180 cycles (Equation 4.5). In Figure 4.16, our experiments show an average performance improvement of 23.5% with both the techniques combined.

With respect to the cost, increasing the L2 banks would require higher number of ports in the crossbar. As the area of a crossbar increases polynomially on increasing the ports, the area overhead will be significant. Energy demands also increase significantly as each router is more complex and need to arbitrate on higher number of nodes. In contrast, CCN only require simple multiplexers at each router and scales well with respect to area and energy overheads. Alternatively, increasing the L2 data path width to provide more L2 bandwidth would also be area intensive as it entails increasing the area of 15×12 core-to-L2 connections in the crossbar, making the crossbar much bulkier. In contrast, CCN only requires 15 core-to-core connections. As core-to-L2 connections are typically longer (in addition to being higher) than core-to-core connections in CCN, there is a higher overhead in scaling the former.

4.6.2 Sharing Tracker

Tarjan and Skadron [149] proposed a scheme to exploit reuse within the private caches by using a Sharing Tracker, a decomposed version of the coherence directory. It aims to reduce the *off-chip memory bandwidth* demand by diverting DRAM accesses to private caches that contain a shared copy. Although we adopt a similar intuition to reuse shared copies in private caches, our aim is to reduce the bandwidth demand to the *shared cache* (and not the DRAM as in [149]). This is because in recent GPU architectures, exploiting *reuse does not considerably reduce off-chip memory traffic* (as shown in Figure 4.11), and hence a common directory in shared cache is not expected to show any benefit since there are not many off-chip memory accesses that it can avoid. In fact, since accessing and maintaining the sharing tracker in L2 cache adds to the bandwidth demand to L2 without relieving pressure on off-chip bandwidth, it will only exacerbate the problem

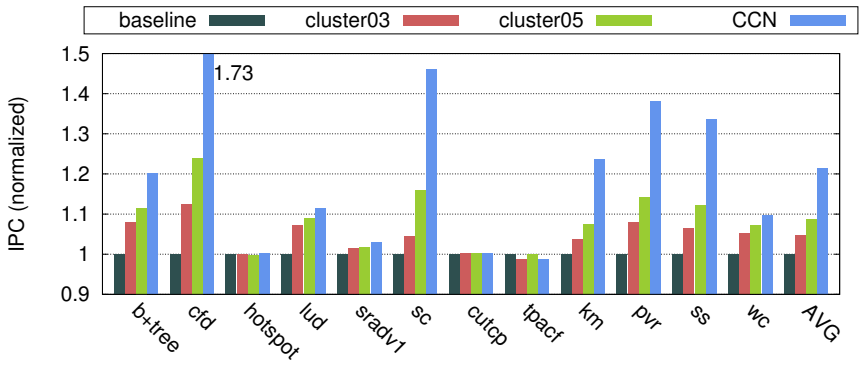


Figure 4.17: Ideal speedup with L1 cache clusters

by increasing L2 access latencies, thereby worsening the performance with respect to baseline. For those architectures where off-chip memory traffic is also reduced by exploiting sharing within private caches, CCN achieves the same, but in addition, it also reduces the traffic to L2 (which we have shown to be critical to performance), and therefore provides a significant advantage over a directory approach.

4.6.3 Clustered Sharing

Keshtegar *et al.* [77] proposed an architecture to enable restricted sharing within core clusters. However, we have shown in Figure 4.2 that while some benchmarks show higher reuse with neighbouring cores, others show a uniform sharing with all cores. In Figure 4.17, we show the ideal performance improvement (with no sharing overheads) obtained by sharing within cache clusters and we compare it with an ideal case of CCN (sharing among all cores). We observe an average performance gain of 4% and 8% with ideal clusters of 3 and 5 L1s respectively, compared to an average performance gain of 21% with ideal CCN. This suggests that for most benchmarks, upon restricting the sharing within cache clusters, SMs lose out on a large portion of reuse possibilities.

Moreover, the cluster-based proposal by Keshtegar *et al.* employs a mesh-type network within a cluster and scales polynomially with the number of cores. Therefore, we expect the area overhead of clusters to exceed the area of ring-based connections in CCN which scales linearly with the number of cores. Furthermore, in current GPUs, SMs are placed linearly around the central L2 cache [115, 116], and therefore, clusters would require longer wires to connect the far-ends of a cluster as compared to only near-neighbour connections in CCN.

4.6.4 Summary

In this section, we have shown that CCN fares well in comparison with alternative techniques. CCN performs better than simply increasing the number of L2 banks while also being partly orthogonal to the latter technique. Sharing tracker is expected to show negative performance gain with the baseline architecture; and restricted sharing within cache clusters significantly reduces the opportunity to capture inter-core reuse.

4.7 Related Work

While sharing across L1 caches is a common occurrence in multiprocessors, as emphasized by the prevalent use of sophisticated coherence infrastructure, we derive significant benefits by exploiting L1 sharing for GPGPU workloads, a property atypical in GPUs. Additionally, in contrast to earlier works [73, 167] where only the off-chip memory bandwidth is considered critical to performance, we identify the criticality of mitigating congestion in the on-chip cache hierarchy between L1 and L2 cache. In the following subsections, we further discuss several prior works related to the ideas presented in CCN and cite their key differences.

4.7.1 Cooperative Caching in CMPs

In the realm of CMPs, Chang and Sohi [23, 24] proposed cooperative caching by adapting the coherence infrastructure. Subsequently, Herrero *et al.* [59] proposed a scalable distributed cooperative caching scheme by re-designing the coherence engine to provide distributed directories. Both schemes aim to provide aggressive latency and capacity benefits for on-chip caches in CMPs. However, since GPUs are relatively more tolerant to latencies, in this chapter we address the problem pertaining to bandwidth in GPUs. In addition, a directory-based scheme is not directly portable to GPUs due to the lack of coherence infrastructure, and therefore our solution proposes an independent lightweight network.

4.7.2 Ring Network

Ring topologies have been used extensively in commercial multiprocessors to provide low cost inter-core communication. Intel's Larrabee [138] employs a bidirectional ring network to allow on-chip communication between *latency-sensitive* CPU cores, coherent

L2 caches and other blocks with each link being 64 bytes wide (net width of 128 bytes). Xeon-Phi [31] also uses bidirectional rings with each ring comprising three independent rings, *viz.*, a 64 bytes data block ring for data transactions, an address/command ring, and an acknowledgement ring for coherence and flow control messages (net width $>$128 bytes). In contrast, CCN enables bidirectional communication between *latency-tolerant* GPU cores by connecting the incoherent L1 caches in a ring. Due to relaxed latency constraints in CCN compared to prior ring interconnects in multiprocessors, the bus width for inter-core transfers is smaller with each link being 8 bytes and 32 bytes wide respectively (net width of 40 bytes). Therefore, our proposal exploits the latency-tolerance property of multithreaded cores to provide low cost inter-core communication through a lightweight ring network.

Furthermore, Campanoni *et al.* [20] proposed a ring cache for HELIX-RC that acts as a distributed first-level cache, preceding the private L1 cache. Each ring node has a cache array to cache shared data and satisfies the loads and stores received from its attached core. To avoid coherence complications, memory addresses are permanently mapped to the nodes of the ring cache. In contrast, each node in the CCN ring network comprises a shadow tag array, needed only for lookups and not for storage of shared data. Subsequent loads to the shared data via CCN are performed directly in the corresponding L1 caches since there is no separate data array for the ring nodes. Therefore, the nodes in the CCN ring network are lighter than nodes in the ring cache proposed in HELIX-RC.

In parallel to the work presented in this chapter [35], Zhao *et al.* [174] proposed *LA-LLC*, a mechanism to exploit inter-core locality by using latent *bidirectional core-to-core connections* in the 2D mesh network, which they consider as a baseline NoC in their study. In their proposal, they maintain sharing information in the L2 cache lines, and redirect requests back to a remote L1 cache if a sharer is detected, via the bidirectional request network. Subsequently, the remote core sends the shared data to the requesting core through core-to-core connections in the response network. However, modern GPUs often employ *unidirectional* networks for request and response paths where core-to-core connections do not exist [141], requiring us to introduce a lightweight ring for core-to-core connections in this work. Furthermore, introducing fully-connected crossbar or mesh routers with core-to-core connections can be expensive with respect to area and energy overheads, as such routers scale polynomially with respect to hardware overheads with increasing connections between nodes.

4.7.3 Shadow Tags

Prior proposals such as Piranha [12] and Niagara [85] have replicated tag structures of the private L1 caches at the shared L2 cache. Such duplicate L1 tags stored centrally in the L2 cache are typically used to construct partial sharing information, thereby reducing indirections to the coherence engine. Duplicate tag structures are also used to reduce redundant write-back traffic to L2 cache from multiple L1s that cache the same shared data. However, in CCN we replicate the tags adjacent to the corresponding L1 caches and do not complicate the L2 cache control. It is used only to prevent deterioration of L1 cache performance due to remote lookups. Moreover, tag updates to shadow tags incur minimum communication overhead in CCN due to physical proximity of L1 caches and shadow tags.

4.7.4 Cache Management

In the field of GPUs, prior proposals such as Sharing Tracker [149] and cluster-based schemes [77] (discussed previously in Section 4.6) exploit reuse within GPU cores via central directory and clustered caches, respectively. Several other schemes have been proposed for GPUs to improve the effective on-chip cache capacity, reduce cache thrashing and improve locality in L1 and L2 caches. Rhu *et al.* [130] proposed a locality-aware memory hierarchy which adaptively adjusts the memory access granularity to prevent over-fetching, providing better off-chip bandwidth utilization. Furthermore, Li *et al.* [101] proposed a tag-split cache to enable fine storage granularity to improve cache utilization, while keeping a coarse access granularity to avoid excessive cache requests. Tarjan *et al.* [150] proposed a scheme to tolerate memory miss latencies for SIMD cores by masking out threads in a warp which are waiting on data and allowing other threads to continue execution, hence utilizing the idle execution slots. Rogers *et al.* [132] proposed a scheduling policy to minimize cache thrashing by monitoring the lost locality in the L1 data cache. Jia *et al.* [69] presented a taxonomy for memory access locality and proposed a compile-time algorithm to selectively utilize the L1 caches. Narasiman *et al.* [113] proposed large warp architecture and a two-level warp scheduling technique to make effective use of resources on GPU. Jog *et al.* [72] proposed a thread block aware scheduling policy to improve the cache hit rates of L1 cache. Choi *et al.* [30] employed techniques such as write buffering and read bypassing to reduce DRAM traffic and improve the L2 cache utilization, thereby addressing the bandwidth constraint between shared cache and DRAM. There has also been work on cache management

policies for heterogeneous CPU-GPU architectures. Yang *et al.* [166] proposed a CPU-assisted prefetching scheme to improve the GPU memory latencies by localizing the data in the LLC cache. Lee and Kim [93] proposed a TLP-aware cache management policy to effectively utilize the LLC for general-purpose workloads. Broadly, the above cache management proposals focus on reducing the miss rate of independent caches by improving cache utilization. In contrast, CCN reduces the collective bandwidth demand of L1 on L2 by diverting some of the misses to remote L1s, without necessarily reducing miss rate of independent L1 caches. Hence, the above mentioned techniques that reduce the miss rate of individual caches are expected to be orthogonal to our work. Given the severity of the memory bottleneck in GPUs (as indicated by the magnitude of PerfX in Table 4.2), no technique alone solves the entire problem, and hence such orthogonal techniques can be used in conjunction with CCN.

In order to mitigate the severity of cache thrashing, several cache bypassing techniques have also been proposed[1]. In CPUs, Gaur *et al.* [47] proposed a bypass policy to selectively fill the exclusive last-level cache with evicted cache blocks from the higher level. Further, Duong *et al.* [38] proposed a policy to protect reusable cache lines from eviction with a dynamically computed Protected Distance, and bypass the miss requests upon lack of unprotected cache lines in a set. In GPUs, high multithreading and low on-chip cache capacity per thread present additional challenges due to severe cache thrashing. Chen *et al.* [28] proposed a dynamic cache management policy that combines L1 cache bypassing and throttling. In their proposed scheme, warp throttling prevents over-saturation of on-chip cache resources while cache bypassing prevents cache contention, requiring lower number of warps to be throttled in comparison to standalone warp throttling schemes. Li *et al.* [99] proposed a locality-driven cache bypassing scheme that uses reuse frequency in a decoupled and extended tag memory to allow allocation in the data memory for only those cache lines that exhibit high reuse. In summary, these cache bypassing schemes in GPUs improve cache utilization by reducing cache thrashing of individual L1 caches. By preventing eviction of cache lines with high reuse, it helps in eliminating repeated reuse requests from the *same* L1 cache to the L2 cache. However, in our proposed technique, we eliminate the reuse requests from *different* L1 caches to the L2 cache. In other words, cache bypassing techniques reduce *intra-core* reuse requests that access the L2 cache, whereas our proposed technique reduces *inter-core* reuse requests that access the L2 cache. Therefore, we expect

[1]Some of the cache bypassing techniques discussed here have also been discussed in the related work for Chapter 3, but succinctly included here for completeness in the current context.

our proposal to be complimentary to cache bypassing techniques as both techniques help in reducing mutually exclusive set of requests to the L2 cache.

4.7.5 Thread Block Scheduling

Inter-core reuse, as shown in this chapter, arises due to sharing of data across different thread blocks (TB) that are scheduled on different SMs. This reuse can also be captured by scheduling such thread blocks, that share data, on the same SM. Li *et al.* [102] proposed a software-based scheme to exploit inter-TB locality by manipulating thread block scheduling. In their proposal, they cluster the thread blocks that are expected to exhibit inter-TB locality and schedule them concurrently or consecutively on the same SM, thereby exploiting reuse through the private L1 cache. While their proposal does not require any hardware changes, it limits the thread blocks that can reuse data amongst each other. This is because only a few thread blocks can be scheduled on the same SM due to occupancy constraints of each SM, thereby limiting the extent of inter-TB reuse. In Section 4.6.3, we have already shown that limiting sharing within a cluster of SMs (or a cluster of thread blocks) reduces the opportunities for sharing and can be sub-optimal. Moreover, data within thread blocks on an SM can be shared only through a small L1 data cache, whereas inter-core sharing proposed in this chapter spans across multiple L1 caches, thereby increasing the sharing opportunity due to higher effective on-chip storage that participates in sharing.

Several hardware approaches have also been proposed to alter the thread block scheduling policy to improve performance. Kayiran *et al.* [75] proposed *DYNCTA* to regulate the number of thread blocks available on each SM. DYNCTA modifies the thread-level parallelism in the system at the granularity of thread blocks, on the basis of some observable application characteristics, in order to mitigate resource contention. However, it is not aimed at harvesting inter-TB locality. Lee *et al.* [96] also proposed to reduce the number of thread blocks available on each SM to reduce resource contention. In addition, they also aim to extract inter-TB locality by scheduling consecutive thread blocks on the same SM. It is based on the observation that sequential thread blocks are more likely to share data. As a result, it restricts inter-TB locality to be harnessed only within a handful of consecutive thread blocks that are assigned on the SM. However, as observed in the heatmaps shown in Figure 4.2, reuse is not limited to neighbouring SMs (or neighbouring thread blocks) and shows different patterns across different applications. Therefore, restricted sharing between neighbouring thread blocks is again

expected to be sub-optimal. More recently, Wang *et al.* [155] proposed *LaPerm*, a thread block scheduling mechanism that schedules parent and child thread blocks on the same SM to maximize parent-child locality. Therefore, they target only a special type of inter-TB locality in their scheme, and is more restrictive than CCN.

4.8 Conclusion

In this chapter, we address an inefficiency in the management of L1 caches in GPUs. We show that as a consequence of high L1 miss rates, high traffic to L2 cache leads to a bandwidth bottleneck between L1 and L2, which in turn leads to high L2 access latencies. In memory-intensive applications, multithreading is unable to hide such high latencies, making it critical for performance.

For general-purpose applications, we discover considerable potential for data reuse within the L1 caches. We exploit this opportunity to reduce the miss traffic to the L2 cache, and thereby reduce the L2 cache bandwidth demand. To achieve this, we present a Cooperative Caching Network which services the L1 load misses cooperatively via a lightweight ring network. We show that GPUs can tolerate reuse latencies gracefully up to 80 cycles, and therefore a ring topology appears to be a cost-effective solution, as it allows us to trade-off higher latencies for simplicity and short wires, *i.e.*, lower power consumption and die-area cost. We also use shadow tag memory, adjacent to each L1 data cache, to decouple the local L1 cache performance from remote L1 cache tag lookups. For applications that do not exhibit any inter-core reuse, we detect the lack of sharing at runtime and prevent the L1 miss requests from incurring the CCN overhead, sending them directly to the L2 cache. For applications that exhibit reuse, our technique improves the IPC by 14.7% while being neutral to applications that show little or no reuse. We likewise reduce the traffic to L2 cache by 29%, and reduce the average memory latency by 24%. As a result, we reduce the total core stall cycles by 26%. Alongside the above improvements, CCN presents an area and energy overhead of 1.3% and 2.5% respectively, and compares favourably with alternative techniques that address the bandwidth issue.

Chapter 5

Managing Thread-level Parallelism

GPUs are increasingly used in upcoming application domains such as Deep Learning, Autonomous Driving and Medical Diagnosis. In such emerging applications, high degrees of thread-level parallelism (via multithreading) are normally required. However, the consequent increase in demand for memory resources, such as caches and memory bandwidth, gives rise to problems such as cache thrashing [66, 67] and bandwidth bottlenecks [36]. Mitigating the above problems by reducing multithreading, however, comes at the cost of restricting parallelism. Due to this tension between *thread-level parallelism* (TLP) and *memory system performance*, balancing the two properties to maximize system throughput poses a significant challenge in GPUs.

5.1 Overview

In GPUs, L1 data cache is one of the most scarce memory resources, and is therefore most vulnerable to cache thrashing. To improve the L1 caching efficiency, several warp scheduling techniques have been proposed that limit the degrees of multithreading. For instance, warp throttling techniques [132, 133] have been effective in mitigating cache thrashing by limiting the number of warps that can execute on an SM at a given time. However, by restricting parallelism, shared system resources such as on-chip interconnects and DRAM may become underutilized. Previously, to address the above limitations, *Priority-based Cache Allocation* (PCAL) [100] has been proposed, classifying warps into two categories, which are referred in this chapter as follows:

1. *Monitored warps*: Warps that are needed to maintain a sufficient degree of parallelism in the system, without being directly concerned about cache performance.

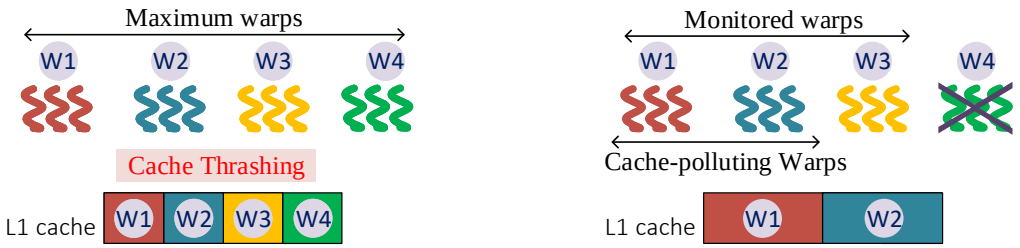


Figure 5.1: Cache footprint with maximum warps and reduced warps

2. *Cache-polluting warps*: A smaller subset of the above monitored warps that are allowed to make allocations and evictions in the L1 cache, in order to maximize cache performance.

The above categorization is depicted in Figure 5.1. In summary, this categorization provides a set of two *knobs* that can be used to fine-tune TLP and memory system performance, thereby reducing cache thrashing and alleviating bandwidth bottlenecks. However, in order to arrive at the optimal number of monitored warps, PCAL employs traditional methods such as iterative hill climbing (detailed discussion in Section 5.3). Such an approach suffers from two major limitations: firstly, hill climbing techniques are susceptible to local optima; and secondly, iterative search can take a long time to converge, particularly in hardware. For these reasons, PCAL is limited in its effectiveness.

Goal: The key goal of this work is to find a good balance between TLP and memory system performance, and to do so expeditiously in hardware. To achieve this balance, we borrow the basic intuition from PCAL to classify warps into two categories, *i.e.*, monitored warps and cache-polluting warps, and use them as knobs to maximize throughput. However, we depart from previous search techniques to find a good composition of warps in each category, overcoming the shortcomings of prior proposals. Note that throughout this chapter, we refer to this dual category of warps as a *warp-tuple*.

Proposal: In this chapter, we propose *Poise*, a different approach to balance TLP and memory system performance via the warp scheduling mechanism. *Poise* comprises two major components — a statistical machine learning framework and a hardware inference engine. The machine learning framework uses a supervised learning model, that is trained offline on a set of profiled kernels using sample input-output pairs. The training set comprises the warp-tuple that resulted in the best performance for a kernel (the output), and the corresponding set of architectural and application features of that

kernel (the input). The input features are carefully chosen using a detailed analytical model. Thereafter, we use a regression model to learn a mapping from the selected architectural and application features, to the chosen warp-tuple. The learned mapping is provided to the hardware via the software or the compiler.

At runtime, the hardware inference engine samples the architectural and application features using hardware performance counters, and uses the mapping that was learned during training to dynamically predict the chosen warp-tuple. To safeguard against statistical errors in prediction, the inference engine performs a focused heuristic search in the near vicinity of the prediction to find a better warp-tuple, if any. This adds resiliency to *Poise* against minor statistical errors arising from the machine learning framework. The final warp-tuple is used by the warp scheduler to alter the number of monitored warps and cache-polluting warps, in order to improve memory system performance, while also maximizing system throughput.

Across a set of benchmarks that were unseen during training, *Poise* achieves a harmonic mean speedup of 46.6% (up to 2.94×) over the baseline greedy-then-oldest (GTO) warp scheduler that employs maximum number of warps. It also outperforms the prior state-of-the-art warp scheduler, PCAL, by an average of 15.1%.

Organization: The remainder of this chapter is organized as follows. Section 5.2 provides the necessary background relevant to this work. Section 5.3 motivates the problem of balancing thread-level parallelism and memory system performance in GPUs and illustrates the shortcomings of prior techniques. Section 5.4 presents a system-level overview of our proposed *Poise* mechanism. Section 5.5 presents the machine learning framework in form of an analytical model and a regression model. The analytical model is used to reveal the most representative architectural and application features that should be used for learning. The regression model learns a mapping from the revealed application and architectural features to the warp-tuples that yield best performance. Section 5.6 presents a hardware inference engine to predict good warp-tuples based on the observed runtime features and the learned mapping, which is used to alter scheduling decisions dynamically. Section 5.7 evaluates the proposed mechanism and presents the results. Section 5.8 presents the related work and Section 5.9 concludes the work by summarizing our key contributions and results.

Table 5.1: Baseline architecture parameters for GPGPU-Sim

Parameter	*Value*
SMs	32
Clock frequency	Core @ 1.4 GHz; Crossbar/L2 @ 700 MHz
Schedulers per SM	2, greedy-then-oldest (GTO) scheduler
Max warps per SM	48 (24 per scheduler)
Max threads per SM	1536
SIMD width	32
Registers per SM	32768
Shared Memory	48 KB
L1 Data Cache	16KB, 128B line, 4-way, LRU, Hash Set-Indexed, 32 MSHR entries
Interconnect	32×24 Crossbar, Fly-topology, 32B flit
L2 Cache	2 MB, 24 banks, 8-way, 128-byte line, LRU
DRAM	GDDR5 DRAM @ 924 MHz, 6 Memory Partitions, 384 bits buswidth

5.2 Background

In this section, we provide the necessary background for this chapter with a brief discussion about the baseline architecture and supervised learning.

5.2.1 Baseline Architecture

In this study, we consider a baseline modelled on a modern GPU, comprised of 32 SMs, 16 KB L1 data cache and 2 MB L2 cache. Each SM can support up to 1536 concurrent threads and up to 48 warps. There are 2 warp schedulers per SM for high throughput issue. Each warp scheduler equally distributes the scheduling load and manages a maximum of 24 warps each at any given time. The baseline parameters are summarized in Table 5.1.

5.2.2 Supervised Learning

Supervised learning is a machine learning technique, which uses a *training set* comprising a set of input-output pairs, and constructs a mapping from the input to the output by analysing the training data. The learned mapping is used to make predictions or *inferences* about the output on unobserved input data. The set of input variables

that are used for training are often referred to as the *feature vector*. In this work, the input feature vector comprises a set of observable properties of the architecture and application; and the output comprises the warp-tuple that results in the optimal (or near-optimal) performance for a kernel (discussed in Section 5.5.2).

Feature Selection: The accuracy of the model depends highly on the selection of the feature vector. While correlation techniques [54, 22] and genetic algorithms [3, 92] are commonly used for selecting a set of representative features, domain knowledge can be leveraged by constructing robust theoretical models [125, 16] to discover a reliable set of features (as shown in Section 5.5.1). This can help reduce the dimensionality of the feature vector to truly representative features and significantly improve the prediction accuracy.

Regression Analysis: Supervised learning algorithms such as Generalized Linear Models (GLM) borrow heavily from statistics [43, 44], and are often referred to as *statistical machine learning* algorithms [90, 19]. In this work, we use Negative Binomial regression [117] from the family of Generalized Linear Models. In this regression model, the output follows a negative binomial distribution. The learned mapping from the input to the output is expressed through a set of *feature weights* for each corresponding input in the feature vector. The logarithm of the output is expressed as a *weighted* sum of input features (as shown in Section 5.5.4).

5.3 Motivation

Since the wide adoption of on-chip memory hierarchies in GPUs [49, 116], several warp scheduling techniques have been proposed to maximize cache performance. In this section, we discuss two prior state-of-the-art techniques and analyse their limitations. Consequently, we motivate the need for an alternative approach that addresses prior shortcomings. We discuss other related and orthogonal techniques in Section 5.8.

5.3.1 Cache-Conscious Wavefront Scheduling

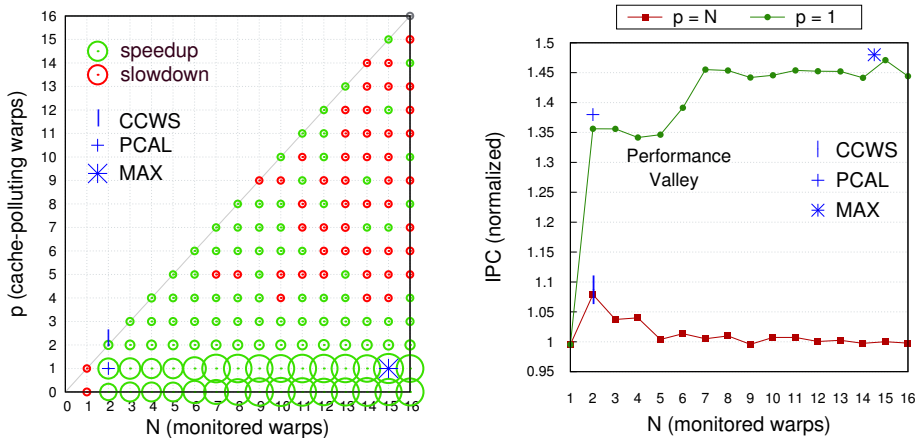
Rogers *et al.* [132] proposed Cache-Conscious Wavefront Scheduling (CCWS), a warp throttling technique to adaptively limit the number of warps, thereby reducing cache thrashing. In their scheme, they maintain Lost Locality Score (LLS) for each warp that indicates the potential cache hits lost by each warp due to cache thrashing. This is done by employing a victim tag array (VTA), which is a modified version of the victim cache.

VTA stores the tag and warp information for the cache lines evicted by the L1 cache. For every L1 miss, the VTA is probed and the LLS is incremented if the load request hits in the VTA. Warps with higher LLS, above a cumulative cutoff, are given exclusive access to the L1 cache. The remaining warps with lower LLS are prevented from issuing load instructions in order to reduce cache thrashing. Due to the high hardware overhead of CCWS, the authors also discuss Static Warp Limiting (SWL), an offline profiling based technique to determine the appropriate extent of throttling for each benchmark. This is done by characterizing each benchmark for every possible level of throttling, and selecting the number of warps that led to best performance. They also note that static SWL outperforms dynamic CCWS due to the runtime overheads of the latter. In SWL, however, every new benchmark needs to be profiled as it does not utilize any prior knowledge acquired from other benchmarks.

5.3.2 Priority-based Cache Allocation

While CCWS successfully improves cache performance by reducing cache thrashing, it severely restricts the thread-level parallelism in the GPU. Li *et al.* [100] observed that throttling also leads to under-utilization of shared system resources. Consequently, they proposed Priority-based Cache Allocation (PCAL) to decouple parallelism and cache performance. In their scheme, they classify warps into two categories — referred in this chapter as *monitored warps* and *cache-polluting warps*. Firstly, *monitored warps* are a group of warps that are allowed to participate in the overall multithreading and are used to meet the high parallelism needs of an application. Secondly, *cache-polluting warps* are a subset of monitored warps that have full cache privileges and are allowed to make allocations and evictions in the L1 cache. Cache-polluting warps are controlled to maintain satisfactory levels of cache performance. Throughout this chapter, we refer to the number of monitored warps as **"***N***"** and the number of cache-polluting warps as **"***p***"**. PCAL aims to find a balance between TLP and cache performance by varying N and p.

In the proposed scheme, PCAL starts by employing the CCWS policy to find the right level of initial throttling. Taking the result of CCWS as the starting point, PCAL performs a search in the $\{N, p\}$ solution space. First, p is varied in parallel across different SMs for a specified duration of sampling. Thereafter, different SMs determine the best performing p through parallel voting. The parallel voting mechanism is managed by a finite-state machine and selects the p that led to best performance. The



(a) Navigating the solution space

(b) Performance peaks and valleys

Figure 5.2: Static profiling of ii kernel #112

best performing p chosen through voting is applied to all SMs. The selected number of cache-polluting warps are maintained until some SM suffers a drop in performance that is greater than a predefined threshold, at which point SMs trigger another round of sampling and parallel voting. Next, the number of monitored warps, N, are selected through an iterative search by hill climbing. To achieve this, the PCAL control unit monitors the shared resources and starts increasing N by one warp at a time if the shared resource utilization is below a certain threshold. The monitored warps are increased as long as SMs observe a threshold level of performance improvement in every iteration. Similar to CCWS, the authors propose both static and dynamic flavours of PCAL, and show that static outperforms dynamic, due to the runtime overheads of the latter, as was observed in CCWS.

5.3.3 Pitfalls in Prior Techniques

In Figure 5.2 we show the above techniques in action for a kernel from the *ii* benchmark and analyse the shortcomings of PCAL and CCWS. The simulation methodology is illustrated later in Section 5.7.3. Firstly, Figure 5.2a explores the $\{N, p\}$ solution space by offline profiling of the kernel across the entire region. Here, the x-axis represents the number of monitored warps (N), while the y-axis represents the number of cache-polluting warps (p, where $p \leq N$). The green and red colour of the circles in the graph represents speedup and slowdown respectively, observed for a warp-tuple indicated by coordinates (N, p); whereas, the radius of the circle is proportional to the magnitude of speedup or slowdown. Additionally, Figure 5.2b shows a plot for two specific values

of p from the prior $\{N, p\}$ solution space, *i.e.*, $p = N$ and $p = 1$, and demonstrates the performance variation in each.

As shown in Figure 5.2a, CCWS binds p with N, and thereby takes values only on the diagonal line $p = N$. Consequently, CCWS technique results in a speedup of 7% at $(2,2)$, which is the peak performance point on the diagonal. In contrast, PCAL decouples p from N, and searches the two-dimensional space. To implement this search, PCAL first uses CCWS to arrive at $(2,2)$. Thereafter, it performs a parallel search in p (converging to $p = 1$) and an iterative hill climbing in N (converging to $N = 2$). In effect, PCAL converges to $(2,1)$, resulting in a speedup of 35%. However, we note that the maximum speedup across the entire $\{N, p\}$ region is 45%, observed at $(15,1)$.

While the inefficiency of CCWS is due to its restrictive coupling of N and p, the sub-optimality of PCAL can be explained due to the following reasons. As shown in Figure 5.2b, the hill climbing in $p = 1$ (green line), starting from the CCWS point at $N = 2$ (on the x-axis), gets trapped at a local optimum at $N = 2$ due to a nearby performance valley at $N = 4$. Consequently, PCAL does not transition to the global optimum at $N = 15$. Therefore, when there are multiple performance peaks in the $\{N, p\}$ solution space, as is the case in GPUs [53], PCAL becomes prone to a local optimum point that is nearest to the starting point. Moreover, even in the absence of performance valleys, if the starting point is far from the performance peaks (as is the case in the above example), it would require multiple iterations to converge on a solution. Therefore, adaptive hardware implementations lead to poor results compared to their static counterparts, as was already observed in dynamic CCWS and PCAL.

5.3.4 Summary

In summary, prior techniques are limited in their ability to efficiently span the $\{N, p\}$ solution space and there are two primary reasons for this. Firstly, conventional methods such as hill climbing are prone to local optimums, and therefore lead to sub-optimal solutions. Secondly, dynamic implementation of prior techniques present considerable time and sampling overheads leading to further degradation in the efficiency of these approaches. Therefore, in this work, we propose an alternative technique to arrive at a well-performing warp-tuple $\{N, p\}$, avoiding the shortcomings of prior techniques discussed above.

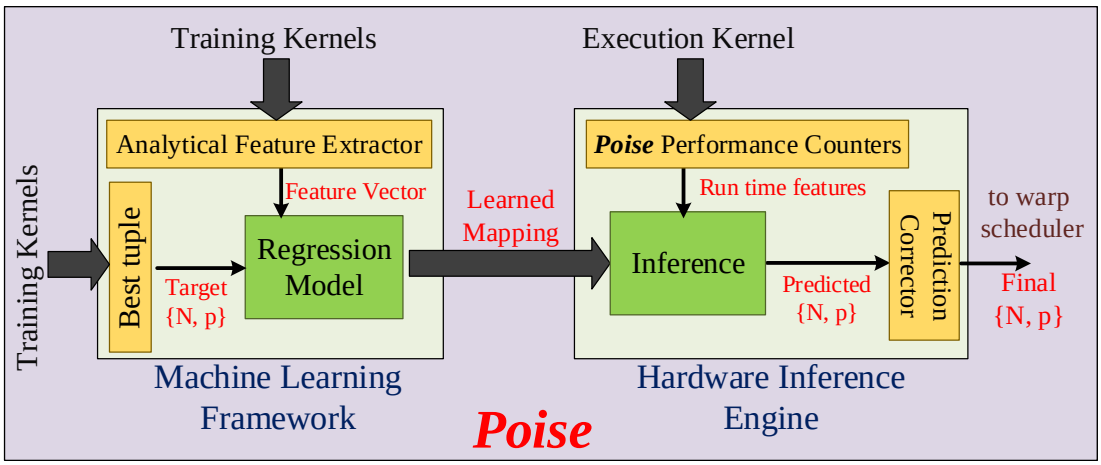


Figure 5.3: System-level architecture of *Poise*

5.4 Poise: A System Overview

We now present *Poise*, a different approach for balancing thread-level parallelism and memory system performance. Figure 5.3 depicts the system-level architecture of *Poise*. It is comprised of the following two major components, which are discussed in detail in subsequent sections.

1. A statistical machine learning framework, where we use a supervised regression model to perform offline training on a set of profiled kernels in the training set. During training, we learn a mapping from a set of application and architectural features to the warp-tuple $\{N, p\}$ that yields best performance.

2. A hardware inference engine, where we sample the runtime features, online, using hardware performance counters, and predict good warp-tuples $\{N, p\}$ for previously unseen kernels, using the mapping that was learned during training. As a result, this strategy drastically reduces the time and overheads involved in finding a good initial solution. Thereafter, we perform a focused heuristic search in the near vicinity of the predicted warp-tuple to find a better warp-tuple, if any, thereby offsetting statistical prediction errors, which are inherent in any machine learning algorithm.

5.5 Machine Learning Framework

In this section, we present our machine learning methodology. We begin by extracting the salient architectural and application features that should be used for training. To

identify such features, correlation techniques are often used in machine learning. However, in computer architecture, we argue for a theoretical exploration of the features to better reason about the accuracy of the developed machine learning framework. It also enables portability across different architectures by providing the theoretical tools to model the variable characteristics. Therefore, we first develop an analytical model to reveal the feature space, and then use those features for training.

5.5.1 Analytical Model

Fundamentally, GPUs employ the following two types of concurrency to hide the long latency of memory accesses. Firstly, via instruction concurrency which is attained by the execution of independent instructions between a memory load and its usage within a warp. Secondly, via warp concurrency which is attained by the execution of independent instructions from other warps, *i.e.*, thread-level parallelism. More specifically, when a warp encounters an instruction that is dependent on a pending load, it is replaced with another warp that has a stream of independent instructions. Thus, these two mechanisms help in keeping the functional units busy when there is sufficient independent work within or across warps [83, 124]. The importance of considering both sources of concurrency mentioned above, *i.e.*, warp concurrency and instruction concurrency, has been emphasized in recent work [153].

In an application, if a typical load and its use are not separated by sufficient independent instructions from the same warp (low instruction concurrency), then higher TLP is required in order to hide latencies (high warp concurrency). However, owing to practical limits on number of warps, each warp would quickly arrive at the dependent instruction and wait for pending memory loads to complete. Therefore, in such applications, load miss latencies determine when the dependencies within a warp can be resolved and appear in the critical path. Such applications are referred to as *memory-sensitive applications*, where improving the memory system performance is more useful than simply increasing the number of warps, as the latter has limited benefit due to a lack of independent instructions. Therefore, instead of operating at the maximum number of warps, memory-sensitive applications require a sophisticated balance between TLP and memory system performance.

In this section, the overarching goal of the analytical model is to mathematically depict the memory latencies that appear in the critical path for memory-sensitive applications and lead to stalls. The initial formulation in our analytical model borrows

from standardized prior work in this area. Hong and Kim [62] proposed an analytical framework called *MWP-CWP*. In their model, they formulate the execution time based on different architectural and application metrics such as memory-level parallelism and instruction-level parallelism. Later, Sim *et al.* [140] proposed *GPUPerf* that comprised of an analytical model extending the MWP-CWP model. In their work, they capture the cache effects using the prior AMAT model [58] and include other optimizations such as instruction mix and memory-level parallelism. We include the lessons from the above analytical models with appropriate changes and reasonable simplifications to tailor it for memory-sensitive applications. Later, using the initial formulation as the starting point, we analyse how the stall cycles are impacted upon varying the number of monitored warps and the number of cache-polluting warps. Finally, we extract the key observable parameters deduced from the analysis and use them to train a regression model.

Maximum warps: To model the miss latencies in a baseline system with maximum warps N, let m_o be the average L1 miss rate on an SM. Furthermore, let L_o be the average memory latency for an individual L1 miss request. Then, upon executing a load instruction concurrently across N warps on an SM, the effective memory latency for the load miss can be expressed by T_{mem} through Equation 5.1. Here, K_{mshr} is the number of MSHR entries in the L1 cache and accounts for memory-level parallelism. Note that we assume each warp instruction generates a single, highly coalesced memory request. Also, the *ceil* function indicates that the effective latency grows as integer multiples of L_o.

$$T_{mem} = L_o \times \left\lceil \frac{N \times m_o}{K_{mshr}} \right\rceil \tag{5.1}$$

$$T_{busy} = N \times h_o \times I_d \times T_{pipe} \tag{5.2}$$

$$T_{stall} = \max\{T_{mem} - T_{busy},\ 0\} \tag{5.3}$$

Next, we model the available slack on an SM to hide the effective memory latency. Let h_o $(= 1 - m_o)$ be the average L1 hit rate for an SM. These L1 hits enable the warps to make forward progress on *dependent instructions* (due to resolved data dependencies), thereby contributing to the busy cycles on the SM. Let I_d be the number of additional instructions in a warp that are now eligible for execution due to a cache hit, until it encounters the next dependency hazard and stalls the warp again. Then the cycles for which the functional units on an SM are kept busy can be expressed by T_{busy} through Equation 5.2. Here, T_{pipe} is the average number of cycles for pipelined execution of a

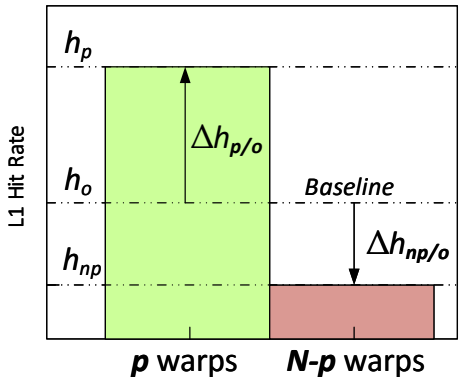


Figure 5.4: L1 hit rate comparison for p and $N-p$ warps

warp instruction on the corresponding functional units. It is noteworthy that the independent instructions from a warp (irrespective of a load hit or a miss) contribute to the parallelism on an SM and keep the functional units busy. However, as mentioned earlier, memory-sensitive applications have only a few independent instructions, allowing us to ignore its effect for simplicity. Finally, the number of stall cycles on an SM when the high latency of memory operations get exposed and appear in the critical path, can be expressed by T_{stall} in Equation 5.3.

Reduced warps: We now consider a scenario when only a subset of warps, p ($\leq N$), can pollute the L1 cache, while the remaining $(N-p)$ warps can only reuse the cache lines allocated by the p cache-polluting warps. In a general case, p warps experience an improved L1 hit rate of h_p while the remaining $(N-p)$ non-polluting warps experience a reduced hit rate of h_{np}. This is shown diagrammatically in Figure 5.4. In such a case, the effective memory latency for concurrent misses across N warps, for a load instruction, can be expressed by T'_{mem} through Equation 5.4, where $m_p = 1 - h_p$ and $m_{np} = 1 - h_{np}$. Note that L' denotes the new average memory latency due to a different level of congestion in the memory system, emerging from the change in the overall L1 miss rate. Similarly, the number of cycles when the functional units on the SM are busy doing useful work, can be expressed by T'_{busy} through Equation 5.5. Therefore, the number of stall cycles in this case can be expressed by T'_{stall} through Equation 5.6.

$$T'_{mem} = L' \times \left\lceil \frac{m_{np}\ (N-p) + \ m_p\ p}{K_{mshr}} \right\rceil \tag{5.4}$$

$$T'_{busy} = \{\ p\ h_p + \ (N-p)\ h_{np}\ \}\ I_d\ T_{pipe} \tag{5.5}$$

$$T'_{stall} = \max\{T'_{mem} - T'_{busy}, 0\} \tag{5.6}$$

Speedup criteria: For a warp-tuple $\{N,\ p\}$ to result in speedup, the resultant stall

cycles must be lower than the baseline scheme. Therefore, using the above equations, the criteria for speedup can be expressed through Equation 5.7.

$$\left.\begin{aligned} T'_{stall} < T_{stall} \implies \tfrac{\Delta T_{busy}}{\Delta T_{mem}} > 1 \end{aligned}\right\} \text{Criteria for speedup}$$
$$\text{where,}$$
$$\begin{aligned} \Delta T_{busy} &= T'_{busy} - T_{busy} \\ \Delta T_{mem} &= T'_{mem} - T_{mem} \end{aligned} \tag{5.7}$$

At this point, we define μ as the *coefficient of goodness* of a warp-tuple $\{N, p\}$ in reducing the stalls cycles compared to the baseline. A higher μ leads to lower stalls, in turn leading to better performance. Using Equation 5.7, μ can be mathematically defined through Equation 5.8.

$$\mu = \frac{\Delta T_{busy}}{\Delta T_{mem}} \implies \text{For speedup,} \quad \mu > 1 \tag{5.8}$$

On further simplification, μ can be expressed through Equation 5.9 using Equations 5.1–5.6. Note that we drop the *ceil* function in ΔT^X_{mem} for simplicity, without significant loss in accuracy.

$$\mu = \frac{\Delta T^p_{busy} + \Delta T^{np}_{busy}}{\Delta T^{np}_{mem} + \Delta T^p_{mem}}$$
$$\text{where,}$$
$$\left.\begin{aligned} \Delta T^X_{busy} &= x\ (h_x - h_o)\ I_d\ T_{pipe} \\ \Delta T^X_{mem} &= \tfrac{1}{K_{mshr}}\ x\ (m_x\ L' - m_o\ L_o) \end{aligned}\right\} \begin{aligned} X &\in \{\text{p, np}\} \\ x &\in \{\text{p, N-p}\} \end{aligned} \tag{5.9}$$

To ensure performance improvement for a warp-tuple $\{N, p\}$, the criteria for speedup given by $\mu > 1$, can be met conservatively if both conditions in Equation 5.10 are met.

$$\mu_{p/np} = \frac{\Delta T^p_{busy}}{\Delta T^{np}_{mem}} > 1 \qquad \mu_{np/p} = \frac{\Delta T^{np}_{busy}}{\Delta T^p_{mem}} > 1 \tag{5.10}$$

$$\mu_{p/np} = T_{pipe} K_{mshr} \left(\frac{p}{N-p}\right) \left(\frac{I_d\ \Delta h_{p/o}}{m_{np}\ L' - m_o\ L_o}\right) \tag{5.11}$$

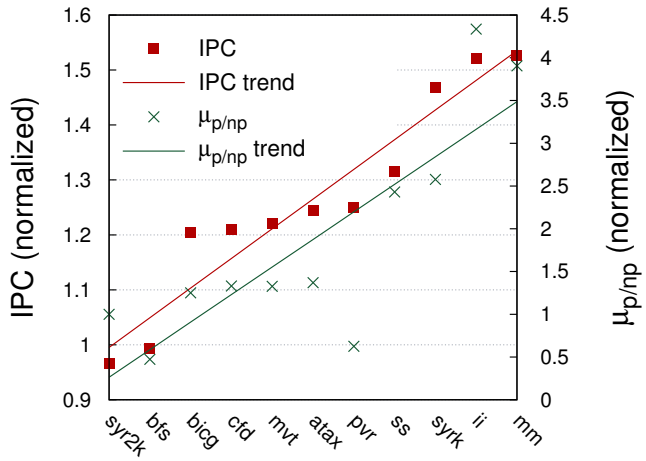


Figure 5.5: Correlation between speedup and $\mu_{p/np}$ with $p = 1$ and $N = 24$

On simplifying, we can represent $\mu_{p/np}$ through Equation 5.11 where $\Delta h_{p/o}$ is $(h_p - h_o)$. Due to symmetrical nature of $\mu_{np/p}$, it is expected to yield similar proportionality as $\mu_{p/np}$ and is therefore omitted for brevity. Therefore, we define $\mu_{p/np}$ as the *objective function* that we wish to maximize for a bivariate warp-tuple $\{N, p\}$.

Correlation with Speedup: In order to measure the correctness of our analytical model, we measure the correlation between the objective function ($\mu_{p/np}$) and the speedup across different memory-sensitive applications. The expectation is that benchmarks with higher values of $\mu_{p/np}$ should demonstrate higher speedups. For the purpose of this evaluation, we select a warp-tuple with $N = 24$ and $p = 1$, and compare the speedup against a baseline configuration with $N = p = 24$. In Figure 5.5, we show IPC normalized to baseline on the y-axis, and benchmarks sorted in the ascending order of speedup on the x-axis. We also show the value of $\mu_{p/np}$ (normalized to $syr2k$) and observe an upward trajectory similar to speedup, albeit non-linear. Therefore, in addition to the actual data points, we also show linear trends in IPC and $\mu_{p/np}$ by plotting the respective data points linearly using linear regression [157]. We observe similar upward trends for both IPC and $\mu_{p/np}$. To quantify the correlation, we measure Pearson correlation coefficient and observe a high value of 0.87 (out of 1.0) between the actual data points of the objective function and the speedup for benchmarks in Figure 5.5, indicating a strong positive correlation between the two metrics. In summary, we observe good correlation between performance speedup and $\mu_{p/np}$. Therefore, we expect Equation 5.11 to be a reliable approximation of our objective function.

Limitations of the Analytical Model: In this work, the intent of the analytical model is not to intricately model the GPU but only to expose the relevant features

by modelling steady-state first-order behaviour. This is because we are ultimately interested in identifying the proportionalities for regression, and not in solving the objective function for exact solutions — allowing room for reasonable simplifications. In that spirit, the model presented in this section is a simplistic approximation of latency hiding in GPUs. For instance, we do not account for dependency latencies due to pending arithmetic instructions. Such latencies would also need to be hidden by parallelism to reduce stall time, just like memory latencies. However, we focus only on the stall cycles contributed by memory latencies as our modifications pertain to memory system performance (and not arithmetic performance), thereby impacting only memory latencies (and not arithmetic latencies). Therefore, it is eliminated for brevity, but can be easily included for completeness.

Similarly, we assume highly coalesced memory access patterns with minimal memory divergence. In case of high memory divergence, the number of memory requests would increase the burden on memory bandwidth, thereby reducing memory-level parallelism due to bandwidth bottlenecks. This would lead to an increase in average memory access latencies. In our model, we already consider memory access latencies observed in a *congested* memory system (L_o and L'), and not the minimum memory access latencies in an uncongested memory system. Therefore, the consequence of high memory divergence is captured indirectly due to an increase in L_o and L' with increasing congestion.

Finally, we execute PTX level code due to the limitations of the simulation infrastructure. A lower level SASS execution would include the effects of compiler optimizations such as instruction reordering, which may alter the number of independent instructions in the dependency chains. This is a limitation of our analytical model. However, the impact of this limitation is expected to be less for memory-sensitive applications where the independent instructions are few.

5.5.2 Feature Vector Representation

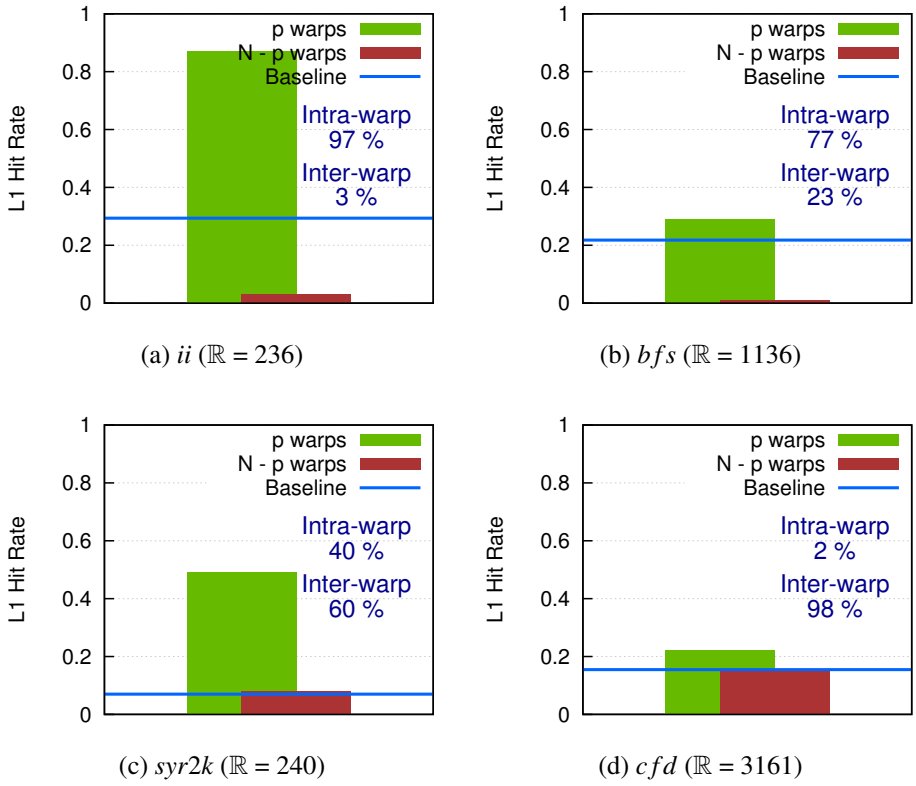
The accuracy of a machine learning model depends highly on the feature vector representation. A low dimensional feature vector with truly representative features is most desirable, as it increases the reliability of the predictions made by the model. Therefore, we leverage domain knowledge to construct a feature vector, through the analytical model discussed in Section 5.5.1. As we have observed that our objective function reliably represents performance, we now extract the salient architectural and

Table 5.2: Variables derived from the analytical model

(a) Objective Function Variables	
Variable	*Description*
T_{pipe}	Cycles for pipelined execution of a warp instruction
K_{mshr}	No. of MSHR entries per L1 cache
h_o	Net L1 hit rate for the baseline system ($= 1 - m_o$)
h_p	L1 hit rate for p warps for $\{N, p\}$ tuple ($= 1 - m_p$)
h_{np}	L1 hit rate for $N - p$ warps for $\{N, p\}$ tuple ($= 1 - m_{np}$)
h'	Net L1 hit rate for $\{N, p\}$ tuple ($= 1 - m'$)
$\Delta h_{p/o}$	Improvement in hit rate for p warps ($= h_p - h_o$)
L_o	Average memory latency for the baseline system
L'	Average memory latency for $\{N, p\}$ tuple
I_d	Average no. of instructions between two different data hazards
(b) Proportionality derived from the Objective Function	
Variable	*Description*
$\mathbb{R}$	Reuse Distance
η_o	Intra-warp hit rate for the baseline system
η'	Intra-warp hit rate for $\{N, p\}$ tuple
$\eta' - \eta_o$	Intra-warp hits that could not be captured initially in the baseline with maximum warps due to cache thrashing
$\Delta h_{p/o}$	Proportional to $\eta' - \eta_o$
$h_o - \eta_o$	Inter-warp hit rate for baseline system
I_n	Average no. of instructions between two global loads ($\sim I_d$)

application characteristics that influence the objective function. To do so, we first make a few observations about the variables that are present in Equation 5.11 and are listed in Table 5.2a. We note that the objective function increases with higher h_p over the baseline h_o (represented by $\Delta h_{p/o}$). Conducive conditions for a high h_p arise when the warps can utilize the cache better in the absence of thrashing. Therefore, there must be enough locality within the warp itself (indicated by *intra-warp locality*) and the footprint of warps must fit in the cache in the absence of thrashing (indicated by *reuse distance*).

We illustrate the above criteria through an example in Figure 5.6 for a warp-tuple configuration where $p = 1$ and $N = 24$. The hit rate for p warps (h_p) is indicated by the green bar; the hit rate for $(N - p)$ warps (h_{np}) is indicated by the red bar; and the hit rate



(a) *ii* ($\mathbb{R}$ = 236) (b) bfs ($\mathbb{R}$ = 1136)

(c) *syr2k* ($\mathbb{R}$ = 240) (d) cfd ($\mathbb{R}$ = 3161)

Figure 5.6: L1 hit rate distribution.

for all warps in baseline system (h_o) is indicated by the blue line. In this figure, we also highlight the different reuse characteristics such as inter-warp hits and intra-warp hits (as a percentage of total L1 hits in the baseline), and reuse distance ($\mathbb{R}$). We observe that *ii* and *syr2k* show a high $\Delta h_{p/o}$. This is explained by the presence of high intra-warp locality (97% and 40% intra-warp hits respectively) and low reuse distance ($\mathbb{R} \leq 240$), presenting enough opportunity to better utilize the cache in the absence of thrashing. However, bfs and cfd have high reuse distance ($\mathbb{R} = 1136$ and 3161 respectively), and therefore, we observe low $\Delta h_{p/o}$ due to continued thrashing caused by the large cache footprint of the warp. Note that if all intra-warp hits are captured in baseline (h_o), then there is no future opportunity to capture more intra-warp hits, despite the favourable reuse characteristics that we have discussed above. Therefore, a good proxy for the remaining opportunity to capture intra-warp locality is the difference between intra-warp hits at $p = 1$ (lowest thrashing) and $p = 24$ (maximum thrashing). A higher remaining opportunity will yield a higher $\Delta h_{p/o}$. We summarize this proportionality between $\Delta h_{p/o}$ and reuse characteristics in Table 5.2b.

Next, we observe in Equation 5.11 that the objective function increases with lower

Table 5.3: Feature Vector (X) and Feature Weights (α; β)

Features: X	Formulation	α (for output N)	β (for output p)
x_1	h_o	0.517687	3.786126
x_2	h'	-0.000261	0.483576
x_3	η_o	7.209138	-6.386444
x_4	η'	-5.977480	10.320107
x_5	$(\eta' - \eta_o)^2$	-8.906397	-6.533500
x_6	$I_n(\eta' - \eta_o)^2$	1.976725	-0.900944
x_7	$(L'm' - m_oL_o)^2/10^5$	0.004668	0.079856
x_8	1 *(constant intercept)*	1.667111	-2.189887

degradation in hit rate for $(N - p)$ warps (indicated by the denominator term). Such a condition arises when the $(N - p)$ warps continue to utilize the cache lines allocated by p warps, despite losing their own ability to allocate and evict cache lines. Therefore, there must be enough locality across warps (indicated by *inter-warp locality)*. In Figure 5.6, we observe that *syr2k* and cfd have high inter-warp hits (60% and 98% respectively), and therefore, $(N - p)$ warps show minimal reduction in hit rate. However, *ii* and bfs have lower inter-warp hits (3% and 23% respectively), and therefore, observe a considerable drop in hit rate for $(N - p)$ warps. Notably, the most favourable conditions for speedup are present for *syr2k*, *i.e.*, high change in hit rate for p warps and low change in hit rate for $(N - p)$ warps.

Finally, we note that I_d can be difficult to compute due to complex data dependency chains. Moreover, in memory-sensitive benchmarks, the different dependent instructions are expected to be in proximity to their preceding load instructions (due to a scarcity of intermediate independent instructions). Therefore, the number of instructions between two dependent instructions (I_d) can be approximated by the number of instructions between two global loads, represented as I_n in Table 5.2b.

Summary: The above analysis revealed several factors that influence the objective function. We summarize the final feature vector X in Table 5.3. The feature weights in the table will be discussed in detail in Section 5.5.4. Note that the polynomial degree for each feature is chosen after sensitivity analysis, in line with general practice in machine learning. Additionally, the variables that depend on the choice of p and N (such as h', η', L' and m_{np}) are measured at a fixed reference point in the two-dimensional $\{N, p\}$ solution space, *i.e.*, (1, 1); the rest are measured at baseline (24, 24). In summary, the

above feature vector X, constructed by sampling at two fixed reference points in the $\{N, p\}$ solution space, provides sufficient substrate to learn about the behaviour of N and p that lead to good performance.

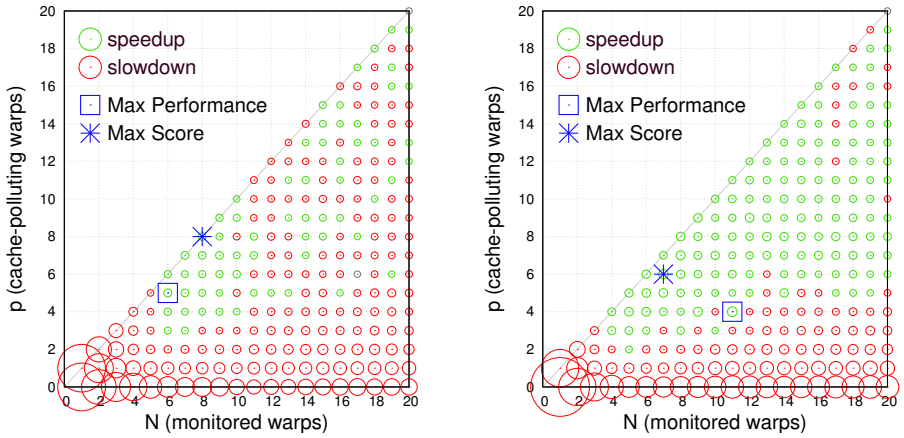
5.5.3 Training Methodology

Scoring Performance Peaks: For supervised learning, we prepare a training dataset that is comprised of profiled kernels from memory-sensitive benchmarks and is used to learn a mapping from the input feature vector X (summarized in Table 5.3) to a target warp-tuple $\{N, p\}$. To select this target warp-tuple, an obvious candidate would be the point in the $\{N, p\}$ solution space that leads to highest performance. However, a prediction error could result in a prediction that is a small distance from the target warp-tuple. Consequently, when the performance peak lies in the vicinity of performance cliffs, a small prediction error can have a negative impact on performance. In such cases, training for a target warp-tuple that lies in a good neighbourhood, even with slightly lower speedup than the global optimum, is expected to yield better results. Therefore, we propose a *scoring system* in which each point in the solution space is assigned a score, which is the weighted sum of performance at the point itself as well as the performance at neighbourhood points.

In our scoring system, the weight assigned to a neighbourhood point is inversely proportional to the distance from the point under evaluation. Therefore, to evaluate the score of a point (a, b) in the solution space, the speedup for that point is assigned a weight of ω_0; the speedup for points at a distance of 1 unit in either N or p are assigned a lower weight of ω_1; and the speedup for diagonal points at a distance of 1 unit in both N and p are assigned a further lower weight of ω_2. Therefore, the score of point (a, b) can be expressed by Equation 5.12, where $S_{x,y}$ represents the speedup at a coordinate (x, y) in the solution space. Consequently, the target warp-tuple for each kernel is the point in the solution space that has the highest score, instead of the point that has the highest performance. Note that the scores are normalized to the number of neighbours, to account for missing neighbours at the boundary points.

$$score(a,b) = \sum_{i \in \{-1,0,1\}} \sum_{j \in \{-1,0,1\}} \omega_{|i|+|j|} S_{a+i,b+j} \tag{5.12}$$

In Figure 5.7, we illustrate the utility of the proposed scoring system by analysing the performance profile of two kernels from the *ii* benchmark profiled across the $\{N,$



(a) Static profile of *ii* kernel#34 (b) Static profile of *ii* kernel#35

Figure 5.7: Scoring performance peaks to avoid cliffs

$p\}$ solution space. In Figure 5.7a, the best performance peak is at (6, 5) resulting in a speedup of 8%. However, at any lower N, performance cliffs start to appear (indicated by red circles). Therefore, the scoring system gives a lower score to (6, 5) due to nearby performance cliffs. Instead, the best score is computed at (8, 8) which presents a safer zone for prediction, even though the target speedup is revised to a lower value of 6%. Similarly, in Figure 5.7b, the performance peak occurs at (11, 4) with a speedup of 15%. However, due to performance cliffs at any lower p or higher N, the best score is instead computed for (7, 6), which presents a slightly lower speedup of 14%, but makes for a safer prediction point. Therefore, such scoring of the performance peaks reduce the likelihood of our target being around performance cliffs so that even with prediction errors, we maintain satisfactory level of performance.

Scaling the Target Warp-tuple: We note that different kernels have different organization of warps, and thereby vary in the maximum number of warps available to the scheduler. Therefore, after obtaining the warp-tuple with best score in the $\{N, p\}$ solution space, we appropriately scale the target N and p to the maximum number of warps that are supported per scheduler, *i.e.*, 24. This ensures uniform bounds for the target warp-tuple in the training data. Later, in the inference stage, we perform appropriate reverse scaling for the predicted warp-tuple.

5.5.4 Regression Model

For regression analysis, we use Negative Binomial regression from the family of Generalized Linear Models (GLM). The rationale for using Negative Binomial regression is

three fold. Firstly, it is used to predict discrete, non-negative target variables, aligning with our requirement for predicting N and p. Secondly, it allows for *overdispersion*, *i.e.*, the variance can exceed the mean of the predicted outcome. This allows more flexibility than the alternate Poisson regression, where the mean is always equal to the variance. Thirdly, it is lightweight due to modest training time and dataset needed to converge to a solution. In contrast, larger models such as Deep Neural Networks are much more computationally intensive, require greater training time and dataset to converge, and are more prone to *overfitting* [151, 89, 144].

$$\ln(N) = \sum_{i=1}^{8} \alpha_i x_i \qquad \ln(p) = \sum_{i=1}^{8} \beta_i x_i \tag{5.13}$$

Using Negative Binomial regression, we construct a log-linear link function to map from the feature vector X, to the target N and p. The link functions can be expressed through Equation 5.13 where x_i belongs to the feature vector X; whereas α_i and β_i are the weights for feature x_i, learned using the regression for N and p, respectively. The learned weights for each feature are summarized in Table 5.3. We evaluate the regression model in Section 5.7.2.

5.6 Hardware Inference Engine

In this section, we present the architecture for *Poise*'s Hardware Inference Engine (HIE). It performs the following two primary functions at runtime: online *prediction* for the chosen warp-tuple, and course *correction* in order to offset any statistical error in predictions.

5.6.1 Prediction Stage

In this stage, HIE dynamically predicts the initial values of N and p that should lead to good performance. To perform such predictions, it requires the feature weights (α and β) that were learned offline during training, and the feature vector (X) that needs to be composed at runtime. At the beginning of kernel execution, the feature weights can be transferred to HIE by the software or the compiler via constant memory. Subsequently, predictions are performed at a periodicity of T_{period} cycles, and this duration is referred to as an *inference epoch*. At the beginning of each inference epoch, HIE reconstructs the feature vector dynamically using hardware performance counters. This is done

by collecting the features listed in Table 5.3 at two locations in the $\{N, p\}$ solution space, *i.e.*, (24, 24) and (1, 1), as was done during training. A modified warp scheduler, discussed in Section 5.6.3, steers the system to each of these warp-tuples for feature reconstruction.

At each of the above two points, HIE performs the following tasks. Firstly, the kernel is executed for T_{warmup} cycles to minimize the crossover effects of changing N and p. Thereafter, performance counters sample the required features for a duration of $T_{feature}$ cycles. Finally, after sampling at both (1, 1) and (24, 24), the link functions (described in Equation 5.13) are used to compute a prediction for N and p. Once the prediction is made, it is appropriately reverse scaled to counter the prior scaling done during training. The final predicted warp-tuple is again fed to the warp scheduler, before moving on to the correction stage. The predictions are reset at the end of each inference epoch or at the end of the kernel, whichever comes first. As an optimization, if I_n is found to be greater than a cut-off I_{max}, then HIE prematurely terminates the inference (and subsequent correction) after sampling at (24, 24). This is to detect compute-intensive kernels that have very few loads (high I_n), and are best run with maximum warps at a warp-tuple (24, 24), due to their insensitivity to cache performance.

5.6.2 Correction Stage

As with any machine learning algorithm, Negative Binomial regression has an inherent error distribution in the prediction outcome. At runtime, we have an opportunity to offset this statistical error and improve the effectiveness of the prediction. Therefore, in this stage, HIE scans the near vicinity of the predicted warp-tuple by performing a neighbourhood search through gradient ascent. This is done by sampling for $T_{correct}$ cycles, after warmup, on either side of the current point at a variable stride (or offset). If the performance at the current location is found to be higher than either neighbours, the stride length is reduced by half. Therefore, as the confidence in the current location increases, the search stride reduces. We terminate the search once the stride length reaches 0. Alternatively, if either neighbour is found to be a higher performance point, the current location is changed to that of the best performing neighbour, and the search is repeated with same stride by searching neighbours around the new location.

In summary, HIE starts by correcting N with an initial stride length of ε_N, while keeping p same as the initial prediction. This is followed by correcting p with an initial stride of ε_p, while keeping N same as the most recently corrected value. After

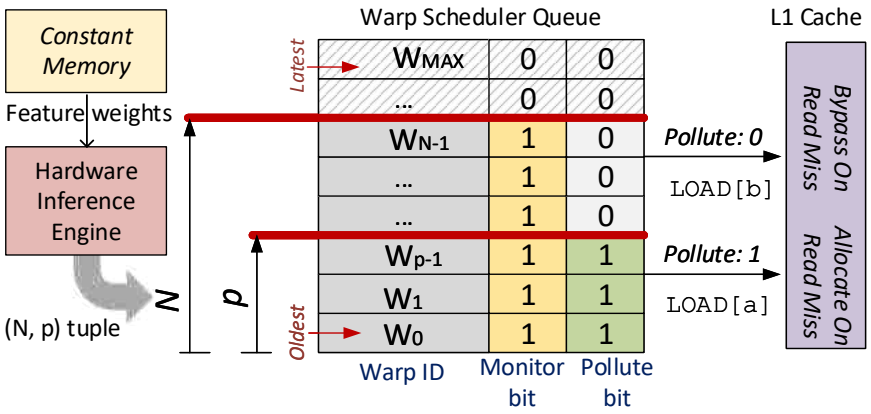


Figure 5.8: Poise Warp Scheduler architecture

converging for both N and p, kernel executes at the final corrected warp-tuple for the remainder of the current inference epoch. It is worth noting that the initial predicted value from the inference stage is likely to be in the near-neighbourhood of the global optimum. Therefore, compared to prior techniques, *Poise* is less likely to get trapped at a local optimum. In addition, the increased likelihood of being in close proximity to higher performing points reduces the overall correction time to arrive at the final solution.

5.6.3 Warp Scheduler

In order to use a warp-tuple $\{N, p\}$ so as to change the number of monitored and cache-polluting warps, we modify the existing GTO warp scheduler. The current scheduler has a queue to track the order in which new warps become active to participate in multithreading. As shown in Figure 5.8, we add an additional *monitor bit* to each entry in the warp scheduler queue, which is set as 1 for N oldest warps. The modified warp scheduler arbitrates (or monitors) only these N warps in a greedy-then-oldest fashion, instead of arbitrating all warps as done in baseline. Furthermore, we also add a *pollute bit*, which is set as 1 for p oldest warps. As done in PCAL, each load request is appended with the pollute bit of the corresponding warp before sending the memory request to the cache hierarchy. On a load miss, the L1 cache-controller uses the pollute bit in the memory request to determine whether to reserve a cache line for the load request or not. Loads without polluting privileges can still access the L1 and incur a cache hit; however, in case of a miss, the corresponding request is forwarded to the L2 without reserving a cache line in the L1.

5.6.4 Summary

As shown in Figure 5.8, the software or the compiler provides the trained feature weights to the HIE via constant memory. During each inference epoch, HIE constructs the feature vector to make a prediction by sampling the relevant performance counters. This requires the warp scheduler to alter the number of monitored and cache-polluting warps, based on the output from HIE at different times. The modified warp scheduler uses the desired N and p values to set the monitor bits and pollute bits in the warp scheduler queue. While the monitor bit determines whether a warp participates in scheduling, the pollute bit determines the privilege of the corresponding load request to reserve cache lines in L1 cache.

5.7 Evaluation

We now discuss our methodology for evaluating *Poise* and demonstrate the results.

5.7.1 Workloads

For the purpose of this study, we use memory-sensitive applications from four major general-purpose benchmark suites, *viz.*, Rodinia [27], MapReduce [57], Polybench [51] and Graph suite [162]. We consider an application as memory-sensitive if the speedup with a 64× larger L1 cache (P_{best}) is greater than 40%. Such benchmarks are listed in Table 5.4, sorted by normalized P_{best}. The benchmarks are split into completely *disjoint* sets for training (3 workloads; 277 kernels) and evaluation (11 workloads; 346 kernels), as is shown in Table 5.4. It is worth noting that the evaluation workloads were unseen during training. We run all benchmarks either to completion or until they execute 4 billion instructions, whichever comes first.

5.7.2 Regression Model Evaluation

We perform the regression analysis using Statsmodels [137], a python-based statistical modelling tool. For the regression, we select only those kernels from the training set that meet certain *threshold* criterion. This is to ensure that training is done on statistically significant data points. For instance, kernels chosen for training must demonstrate at least a threshold level of performance improvement at their target warp-tuple. Furthermore, in our experiments, we observe that infrequent predictions are

Table 5.4: Training and evaluation workloads

#	*Suite*	*Benchmark*	*Abbrv.*	*# Kernels*	P_{best}
Training Set					
1	Graph	Graph Coloring	*gco*	12	3.43
2	MapReduce	Page View Rank	*pvr*	248	2.07
3	Graph	Component Label	*ccl*	17	1.49
Evaluation Set					
1	Polybench	Symmetric rank-2k operations	*syr2k*	1	14.13
2	Polybench	Symmetric rank-k operations	*syrk*	1	9.03
3	MapReduce	Matrix Multiplication	*mm*	23	6.20
4	MapReduce	Inverted Index	*ii*	118	5.94
5	Polybench	Scalar and Vector Multiplication	*gsmv*	2	3.23
6	Polybench	Matrix Vector Product	*mvt*	1	2.97
7	Polybench	BiCGStab Linear Solver	*bicg*	2	2.93
8	MapReduce	Similarity Score	*ss*	164	2.85
9	Polybench	Matrix Transpose	*atax*	2	2.73
10	Rodinia	Breadth-First Search	*bfs*	24	1.55
11	Rodinia	K-Means	*kmeans*	8	1.42

Table 5.5: *Poise* parameters

Parameter	*Description*	*Value*
$\omega_0, \omega_1, \omega_2$	Performance scoring weights	1, 0.50, 0.25
T_{period}	Inference periodicity	200,000 cycles
T_{warmup}	Warmup duration	2,000 cycles
$T_{feature}$	Sampling duration for feature collection	10,000 cycles
$T_{correct}$	Sampling duration for correction	4,000 cycles
I_{max}	Cut-off for instructions between global loads	49
ε_N	Search stride for N	2
ε_p	Search stride for p	4
Threshold speedup	Speedup for training kernels	$\geq 1.5\%$
Threshold cycles	Execution cycles for training kernels at baseline	$\geq$ 10,000 cycles
Threshold hit rate	L1 hit rate for training kernels at $N = 1$, $p = 1$	> 0 %

sufficient to capture most of the performance improvement. Therefore, to prevent frequent predictions during kernel execution, the inference period is intended to be larger in comparison to the time needed to construct a prediction. As a result, we

collect features only for 24,000 cycles (2,000 cycles for warmup and 10,000 cycles for sampling at each of the two points in the solution space). It amounts to only 12% of the entire prediction period (200,000 cycles). Therefore, a good initial prediction accuracy would allow us to find a good performing warp-tuple expeditiously within 12% of the execution time in every inference epoch, followed by the correction phase to further improve performance. The various timing and threshold parameters for *Poise* are derived after detailed sensitivity analysis, and are summarized in Table 5.5.

For a preliminary evaluation of the model, we measure the offline prediction accuracy of the model against unseen profiled kernels from the evaluation set. We observe a mean prediction error of 16% and 26% for N and p, respectively. At runtime, *Poise*'s HIE allows for correcting these prediction errors. Note that due to higher statistical error in p compared with N, the search stride to correct prediction errors in p is chosen to be higher than the search stride for N, *i.e.,* ε_p is 4 and ε_N is 2.

5.7.3 Experimental Methodology

We model a modern GPU on a cycle-accurate simulator, GPGPU-Sim (v3.2.2) [10], based on the architectural parameters listed previously in Table 5.1. For energy and area simulations, we use GPUWattch [98], a McPAT-based power model integrated in GPGPU-Sim. We compare *Poise* with different techniques that are summarized below:

GTO: It represents the baseline greedy-then-oldest warp scheduler, with maximum allowable warps enabled per SM.

SWL: It represents the Static Warp Limiting policy [132] from the CCWS scheduler, which is discussed in Section 5.3.1. In SWL, the optimal number of warps per scheduler are determined through static profiling of benchmarks, which does not incur any runtime overheads. Therefore, our comparison with the CCWS scheduler is conservative in favour of CCWS.

PCAL-SWL: It represents the Priority-based Cache Allocation policy [100] discussed in Section 5.3.2. To determine the initial starting point, SWL (static scheme) is chosen instead of CCWS (dynamic scheme) to eliminate the initial runtime overheads. Therefore, our comparison with the PCAL scheduler is conservative in favour of PCAL.

Static-Best: It represents the configuration when each kernel in an application is run at the best performing warp-tuple. It is determined by offline profiling of all kernels in the $\{N, p\}$ solution space. Therefore, it represents the statically optimal performance of a benchmark derived at *kernel granularity*.

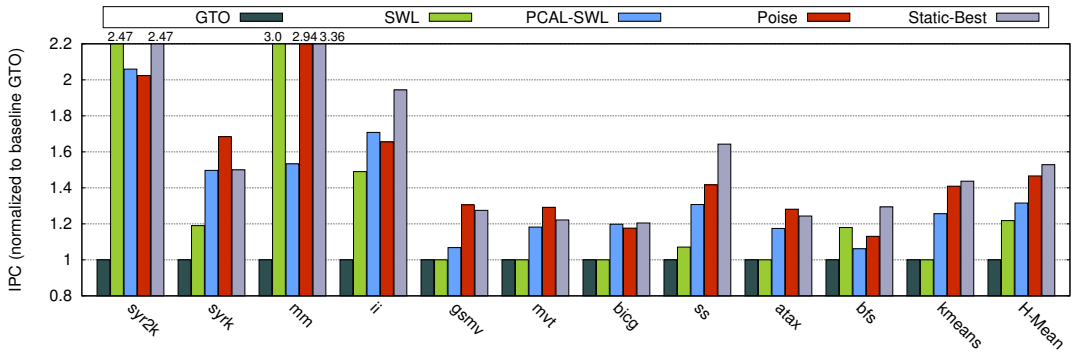


Figure 5.9: Performance normalized to GTO

5.7.4 Performance

In Figure 5.9, we demonstrate the performance of *Poise* normalized to the baseline GTO scheduler for evaluation set workloads. We show that *Poise* achieves a harmonic mean speedup of 46.6% (and up to 2.94× for *mm*). In contrast, we observe a speedup of 31.5% with PCAL-SWL and 21.8% with SWL. Therefore, on average, *Poise* outperforms PCAL-SWL by 15.1% (up to 141.1% for *mm*), and SWL by 24.8% (up to 49.4% for *syrk*). Overall, *Poise* performs better than PCAL because of the following reasons. Firstly, *Poise* is able to predict a good initial warp-tuple expeditiously, using the learned model. Secondly, the near-neighbour search in *Poise* around the predicted warp-tuple is less likely to be trapped at a local optimum as it is expected to be closer to the global optimum, unlike PCAL. We also observe that Static-Best achieves a harmonic mean speedup of 52.8%, surpassing *Poise* only by 6.2%. This performance gap between *Poise* and Static-Best can be attributed to the prediction errors in the regression model, and the slight correction overhead to offset such errors at runtime. Notably, for some benchmarks, such as *atax*, *gsmv*, *mvt* and *syrk*, *Poise* even surpasses the performance of Static-Best. We observe that these applications have monolithic kernels instead of several smaller kernels (as shown in Table 5.4); as a result, *Poise* is able of capture the slight dynamic phase changes within the large monolithic kernels by performing predictions at regular intervals. However, these phases go undetected in Static-Best, where profiling is done at coarse kernel granularity.

Finally, we note that for a few scenarios such as *syr2k* and *bicg*, SWL or PCAL-SWL perform better than *Poise*. This happens when the global optimum lies within (or close to) the narrow reach of the SWL, *i.e.*, the $N = p$ region in the solution space. As both of these schemes use a static SWL profiler, they get a head start by finding (or getting close to) the global optimum without incurring any runtime overheads.

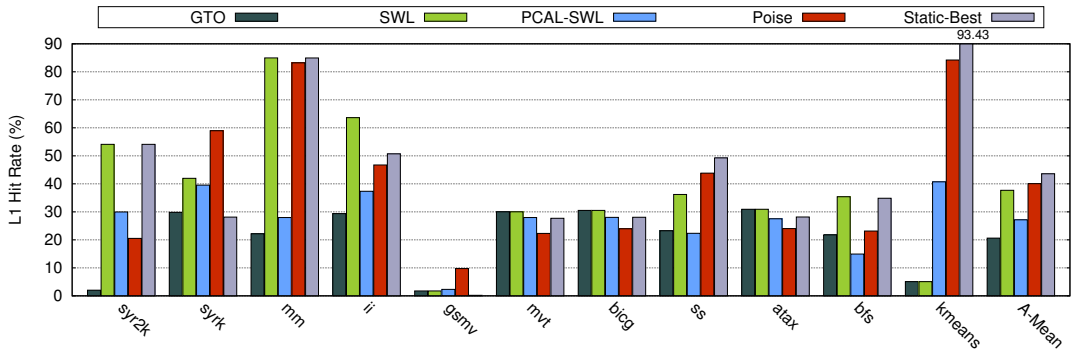


Figure 5.10: Overall L1 hit rate

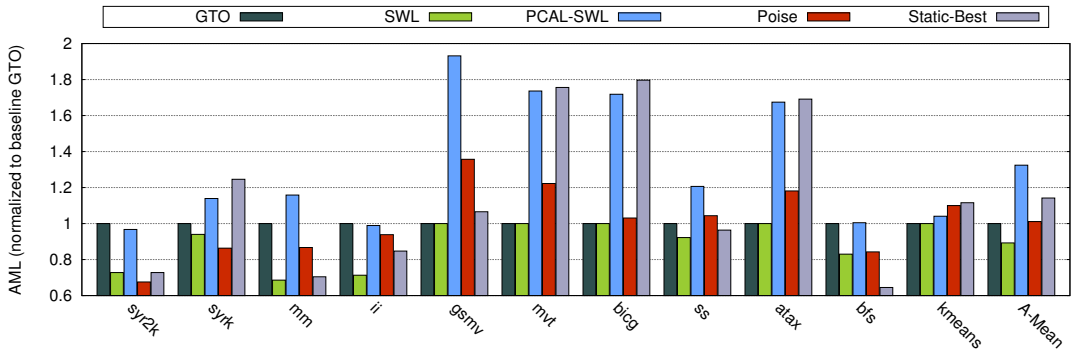


Figure 5.11: Average Memory Latency (AML) normalized to GTO

5.7.5 L1 Cache Hit Rate

In Figure 5.10, we compare the absolute L1 hit rate for different techniques. We observe that *Poise* achieves an average L1 hit rate of 40.1%, in contrast to 27.1% with PCAL-SWL, 37.7% with SWL, and 20.6% with baseline GTO. Therefore, in caching efficiency, *Poise* outperforms PCAL-SWL by 13%, SWL by 2.4%, and GTO by 19.5%. Notably, SWL comes close to *Poise* in L1 hit rate, however, at the cost of significant reduction in system performance. Lastly, *Poise* comes close to the L1 hit rate of 43.6% achieved with Static-Best, indicating the effectiveness of *Poise* in mitigating cache thrashing.

5.7.6 Average Memory Latency

To evaluate the performance of the shared memory system, we measure the average memory latencies (AML) incurred by L1 misses. In Figure 5.11, we observe that *Poise* increases the AML by only 1.1% over the baseline GTO scheduler. In contrast, PCAL-SWL increases the AML by 32.4%. This is because of the lower L1 hit rate in PCAL-SWL compared to *Poise*, which increases the memory traffic and aggravates congestion, thereby leading to high memory latencies. On the other hand, SWL decreases the AML

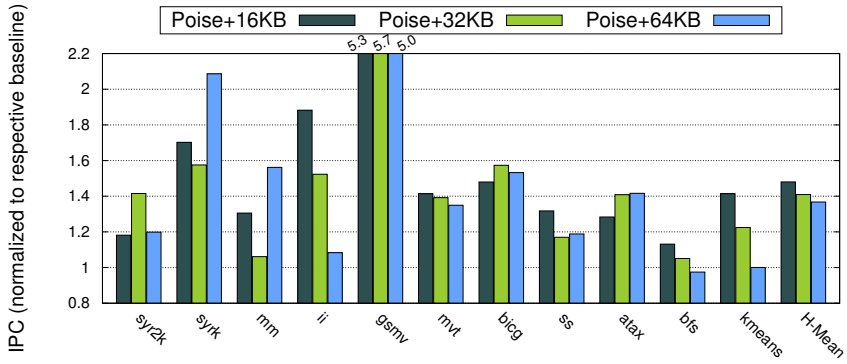


Figure 5.12: Sensitivity to L1 cache size

by 10.7% but significantly underestimates the number of monitored warps, indicated by the low speedup. Interestingly, AML with Static-Best increases by 14.1%, indicating that with optimal warp-tuples, SMs can tolerate a higher AML compared to the baseline.

In summary, we observe that *Poise* provides a good balance between TLP (indicated by speedup) and cache performance (indicated by L1 hit rate), without under-utilizing or over-utilizing shared memory resources (indicated by AML).

5.7.7 Sensitivity Study

L1 Cache Size: The training for *Poise* was performed on a GPU with 16 KB L1 cache, alongside a hash set-indexing function for L1. We now alter the architectural parameters of the *evaluation platform*, while using the previously trained regression model. For evaluation, we employ a linear set-indexing function for L1 and vary the L1 cache size. In Figure 5.12, we observe that with a 16 KB L1 cache, *Poise* maintains a considerable harmonic mean speedup of 48%. Even on increasing the L1 cache size significantly by up to 4× (64 KB), we observe a harmonic mean speedup of 36.7%. Therefore, *Poise* continues to deliver performance improvements even with considerably larger caches. This also highlights the severity of the cache thrashing problem in GPUs. In summary, we observe that *Poise* remains effective even with changes to critical architectural features, such as L1 cache capacity and indexing, despite being trained on a different baseline.

Search Stride: In Figure 5.13, we vary the stride lengths for N and p, represented by (ε_N, ε_p), which are used to perform a neighbourhood search in the correction phase. We note that without any corrections for the predicted values of N and p, *i.e.*, stride of (0, 0), *Poise* achieves a harmonic mean speedup of 23% (up to 3.12×). Therefore, relying

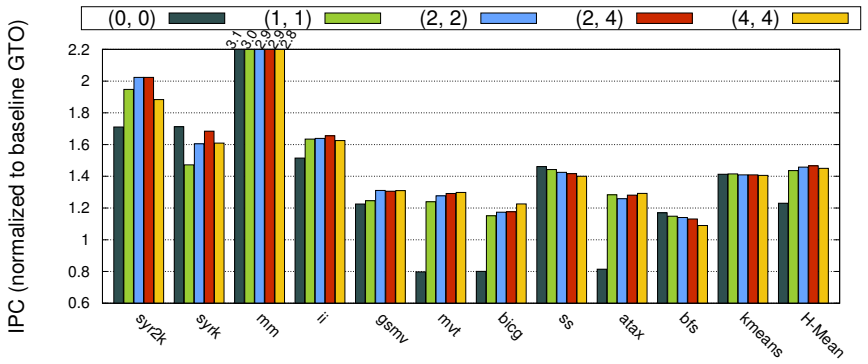


Figure 5.13: Sensitivity to search stride (ε_N, ε_p)

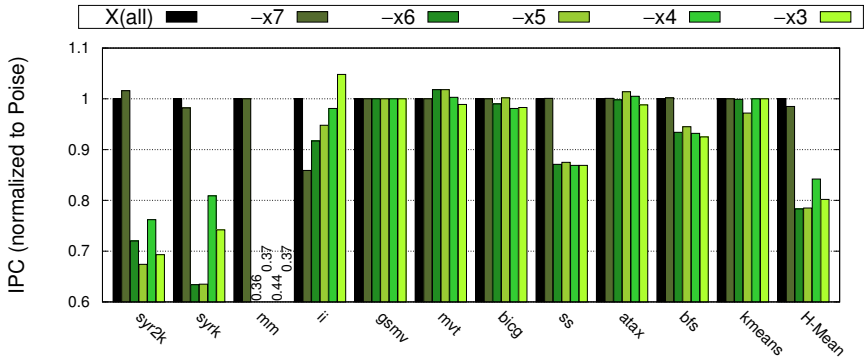
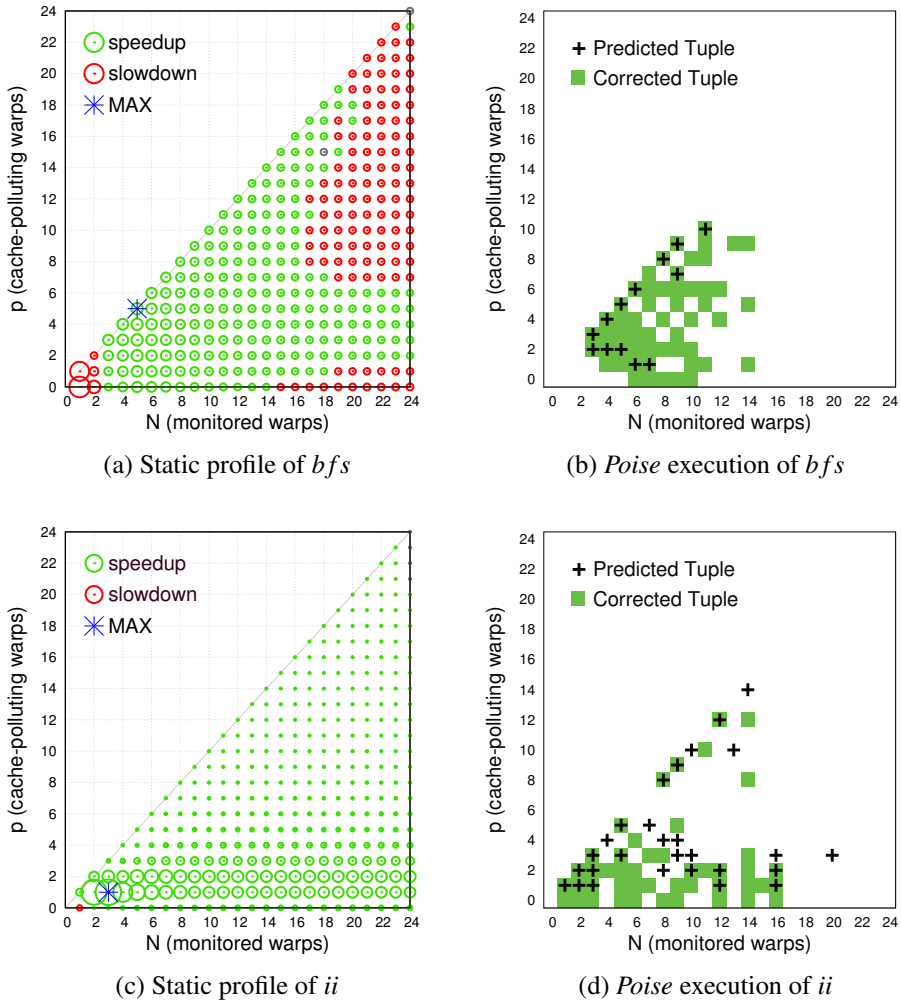


Figure 5.14: Sensitivity to removing a feature x_i from X

purely on predictions, with no correction mechanism, *Poise* still achieves a higher speedup than SWL, while remaining only 8.5% short of PCAL-SWL performance, on average. On increasing the search stride to (1, 1) and (2, 2), we observe the harmonic mean speedup of 43.6% and 45.7% respectively, which settles at 45% for a search stride of (4, 4). Therefore, we note that for most benchmarks, such as *syr2k* and *ii*, increasing the stride length results in improvement at first, but it saturates or wears off with longer strides. However, a stride of (2, 4) gives the best speedup of 46.6% on average. This is due to the lower prediction error in N compared to p (as seen in Section 5.7.2), thereby requiring higher correction only in the latter.

Training Features: We now examine the effect of removing a feature, x_i, from the feature vector X, and retraining the regression model. The resulting execution speedup with such a model is shown in Figure 5.14, normalized to the case when all features are used for training. In each of these cases, no correction is done to the initial predictions, so as to measure the change in actual prediction accuracy. Also, we omit x_1 and x_2 as they are represented in x_7 and show a similar trend. We observe that on removing



(a) Static profile of bfs

(b) *Poise* execution of bfs

(c) Static profile of ii

(d) *Poise* execution of ii

Figure 5.15: Comparing static and *Poise* executions

a feature, the harmonic mean slowdown, compared to the case when all features are used for training, varies from 1.5% for x_7 to 21.7% for x_6. We also note that highly memory-sensitive applications are most adversely impacted by the removal of a feature, emphasizing the importance of the proposed features. In summary, best performance is shown when all features are used for training.

5.7.8 Case Study

We now present a case study for two representative benchmarks, bfs and ii, from the evaluation set of benchmarks. Through this study, we qualitatively illustrate the accuracy of *Poise* in predicting good performing warp-tuples on previously unseen benchmarks. In Figure 5.15a, we show the static performance profile for bfs. It

indicates a general trend suggesting that the speedup improves with lower values of N and p (green circles), with the best performing warp-tuple at (5, 5). It also indicates that there is an aversion to higher values of N and moderate-to-high values of p (red circles). Next, in Figure 5.15b we show the different warp-tuples chosen by *Poise* at runtime during the multiple prediction and correction phases throughout the execution. The predicted warp-tuples are indicated by '+' sign, whereas the warp-tuples generated after correction are indicated through the shaded coordinates. Therefore, we observe that most predictions are in the high performance zone, *i.e.*, in close proximity of the best performing warp-tuple at (5, 5). Furthermore, *Poise* successfully steers the system away from the low performance zones (red circles in Figure 5.15a), thereby correctly detecting the general affinities in a previously unseen benchmark.

Similarly, in Figure 5.15c, we observe that the *ii* benchmark shows an affinity for low values of p and low-to-moderate values of N (indicated by the larger radius of the green circles). The best performing warp-tuple in the static profile is found at (3, 1). In Figure 5.15d, we observe that multiple predictions made by *Poise* at runtime lead to low values of p and low-to-moderate values of N, steering the system to the good performing regions in the solution space. For the prediction errors (indicated by some predictions for higher values of p), *Poise* performs the necessary corrections at runtime, which are traced through the shaded areas in Figure 5.15d.

5.7.9 Hardware Overheads

Poise requires four 32-bit performance counters per SM to collect runtime features such as intra-warp hits, total L1 hits, AML and the number of compute instructions per load instruction. These performance counters provide sufficient substrate to construct the entire feature vector described in Table 5.3. *Poise* also requires arithmetic resources to compute the link function. However, the existing arithmetic units on the SM can be used for this purpose as they are often idle in memory-sensitive applications due to high stall cycles, thereby obviating the need for any extra hardware. Finally, *Poise* requires one finite-state machine (FSM) per SM to manage the transition from prediction stage to correction stage in HIE, and to perform this periodically. In our design, we observe that the FSM requires 7 states for the hardware inference engine, thereby requiring two 3-bit state registers per SM for maintaining the states, in addition to minor combinational logic which can be borrowed from the idle execution units. In total, *Poise* poses a minimal storage overhead of 536 bytes, *i.e.*, less than 0.01% of chip area, estimated

using the existing parameters in McPAT. The minor energy consumption of the above registers is within McPATs margin of error, and therefore, not reported. In summary, *Poise* is extremely lightweight in terms of hardware overhead. In contrast, dynamic PCAL and CCWS implementations require CCWS-like hardware, including victim tag arrays, presenting greater hardware overheads [132, 100].

5.7.10 Discussion

We now briefly discuss few issues pertaining to the applicability of *Poise*.

Portability: In this work, we evaluate *Poise* on a modern GPU. For similar multi-threaded architectures, the feature vector is expected to remain the same, and require only the feature weights to change. Therefore, *Poise* can be adapted to similar multi-threaded architectures by retraining the regression model and generating new feature weights. As these feature weights are provided to the hardware through the software or the compiler, it entails no changes in the hardware.

Need for Poise: With recent support for multi-kernel execution on GPUs, different kernels executing on different SMs can present varied requirement for thread-level parallelism. In such scenarios, the adaptive nature of *Poise* allows it to tailor the scheduling decisions for each SM. This is also useful in multi-chip GPUs [9] or server GPUs [63] where applications with significantly diverse parallelism and memory system needs might be collocated on the same GPU.

5.8 Related Work

In this section, we discuss prior work related to the ideas discussed in this chapter.

5.8.1 Cache Management and Warp Scheduling

In addition to the state-of-the-art warp scheduling techniques discussed in Section 5.3, several cache management schemes have been proposed to improve caching efficiency[1]. Li *et al.* [99] used reuse frequency and reuse distance to bypass the L1 for low locality accesses, using decoupled L1 data and tag arrays. Xie *et al.* [161] proposed locality-driven cache bypassing at the granularity of thread blocks. In contrast to bypassing

[1]The cache management, cache bypassing and warp scheduling techniques discussed here have also been discussed in the related work for prior chapters, but succinctly included here for completeness in the current context.

schemes, we not only improve cache performance, but also alter the levels of multi-threading. Furthermore, Chen *et al.* [28] proposed a coordinated cache bypassing and warp throttling scheme. However, similar to PCAL, they iteratively alter the number of warps by hill climbing to optimize NoC latencies. Therefore, it suffers from the same limitations as PCAL that were discussed in Section 5.3.3. On similar lines, Khairy *et al.* [78] proposed a coarse-grained cache bypassing scheme where the entire L1 or L2 cache is bypassed for all accesses if the cache miss rate exceeds a predefined threshold. They also proposed a dynamic warp throttling scheme using core sampling, where different SMs employ different number of warps during the sampling period and vote for the best level of throttling at the end of the sampling. However, it is a single dimensional warp throttling scheme and suffers from the same limitations as SWL. More recently, Lee and Wu [94] proposed an instruction-based scheme to track low reuse memory instructions and bypass requests from such instructions. Similarly, Koo *et al.* [86] proposed an instruction-based scheme to not only bypass, but also to protect cache lines using instruction locality characteristics. In contrast, *Poise* is a warp-level scheme, and therefore, can be applied orthogonally to instruction-level schemes, given the large magnitude of the problem in GPU caches.

Previously, Rogers *et al.* [133] proposed divergence-aware warp scheduling to capture intra-warp locality in loops. Jia *et al.* [69] presented a taxonomy for memory access locality and proposed a compile-time algorithm to selectively utilize the L1 caches. Narasiman *et al.* [113] proposed large warp architecture to dynamically create warps to mitigate penalty due to control flow divergence. Jog *et al.* [72] proposed a two-level warp scheduling technique to prevent the warps from reaching a long latency memory operations at the same time. Xie *et al.* [161] proposed a framework to extract locality information about the loads, and consequently make cache bypassing decisions to restrict the number of thread blocks that can access the cache. Kim *et al.* [82] proposed to utilize the long latency stalls by pre-executing instructions that are not in the dependency chain of the stalled instructions. Oh *et al.* [119] maximize the utilization of cache lines by prioritizing warps based on their load instruction characteristics. Tarjan *et al.* [150] proposed a scheme to tolerate memory miss latencies for SIMD cores by masking out threads in a warp that are waiting on data and allowing other threads to continue execution, hence utilizing the idle execution slots.

5.8.2 Machine Learning in Systems

In the realm of compilers, machine learning based techniques have been extremely useful in the areas of autotuning and compiler optimizations. Stephenson *et al.* [145] used genetic algorithms to find effective compiler optimizations by searching in the solution space of priority functions (or cost functions). They perform training by populating the model with different expressions of the priority function. Expressions that lead to the best performing code are considered fit and selected for crossovers, thereby populating the next generation of expressions. This methodology helps in generating good application-specific and general-purpose heuristics. Agakov *et al.* [3] proposed a methodology to speedup compiler optimizations for embedded platforms. They use machine learning to focus the search on those areas of the optimization space that are most profitable with respect to performance. This is done by correlating new programs with previously observed programs and using prior information to focus the optimization search. In the above work, authors employ Principal Component Selection [17] to select a set of static code features and use independent distribution and Markov models to perform learning. In a subsequent work, Cavazos *et al.* [21] proposed the use of hardware performance counters to capture dynamic features, instead of static features, to determine good compiler optimizations. They select all hardware performance counters available in the architecture. However, a limitation of the above work is the inability to discover new features that do not natively exist as performance counters, but if included, might be suitable in improving the model. On similar lines, Park *et al.* [123] proposed a machine learning model to learn good polyhedral optimizations using dynamic behaviour of a program observed via hardware performance counters, thereby avoiding an extensive heuristic search. Fursin *et al.* [46] proposed MILEPOST GCC, a self-optimizing compiler based on machine learning to optimize programs for evolving hardware, such as configurable embedded processors. In our work, we address an *architectural* optimization problem. In contrast to the compiler optimization space, the architectural optimization space discussed in our work is much smaller. However, as we perform dynamic optimizations in hardware using runtime features, conventional iterative search techniques still pose a significant runtime penalty, as was shown in Section 5.3.3. Therefore, despite the difference in shape and size of the optimization space, speeding up architectural optimizations proves to be useful in our scenario as well. Furthermore, to alleviate the black box nature of the machine learning framework, we select application and architectural features by harnessing domain knowledge through

an analytical model, instead of using automated feature selectors.

Application of machine learning in computer architecture has been picking up in the recent years. Jiménez and Lin [71] proposed a dynamic branch predictor based on the perceptron — the simplest neural network. In their scheme, they replace the existing two-bit saturating counter with a perceptron predictor. The predictor is trained by learning the correlation between the outcome of prior branches in the global history and the outcome of the current branch. Predictions are made by retrieving the appropriate perceptron weights through the branch address and using the runtime global history register as the input to the perceptron. İpek *et al.* [65] applied reinforcement learning to adaptively change DRAM scheduling decisions, instead of employing rigid scheduling policies. The adaptive memory controller aims to perform optimal actions, under a given system state, that maximizes the long-term reward of those actions. Liao *et al.* [103] used machine learning to optimize memory prefetch decisions in datacenters by detecting the varying application needs through hardware performance counters. Machine learning has also been employed to predict performance and power trends to avoid running full cycle-accurate simulations. İpek *et al.* [64] built a design space model to predict the performance impact of architectural changes, saving considerable simulation time. Lee and Brooks [92] proposed a methodology to discover efficient configurations of reconfigurable microarchitectures over a large adaptive space of microarchitectural parameters. In their methodology, they employ a collection of techniques such as sparse sampling (to minimally navigate the design space), regression-based predictive modelling (to learn about the performance impact of design changes without extensive simulations), and genetic algorithm (for combinatorial optimization of designs). Wu *et al.* [158] used clustering algorithms and machine learning in GPGPUs to estimate power and performance trends, using previously observed scaling behaviours. Bitirgen *et al.* [18] used machine learning to predict coordinated resource allocation decisions for shared resources in a CMP in order to optimize system-level performance.

5.9 Conclusion

In the computer architecture community, the use of machine learning to solve architectural problems has been oddly limited, compared to other fields. Few reasons for this limited use is the bulky nature of sophisticated models such as Deep Neural Networks, that generate prohibitively large feature weight matrices with high storage needs, and present high computational demands for training and inference. These factors make

them difficult to use and adopt in architectures, where on-chip resources are often severely limited. Moreover, a black box nature of these techniques, due to a lack of mathematical models to justify their performance, often makes it difficult for architects to argue about their effectiveness across different architectures and applications.

In this chapter, we propose *Poise*, a combination of machine learning and hardware techniques, to balance thread-level parallelism and memory system performance in GPUs. In the machine learning framework, we present a supervised learning model that is trained offline on a large set of profiled kernels. It learns a mapping from a set of application and architectural features, to good warp scheduling decisions that led to best performance. This learned mapping is provided to the hardware through the software or the compiler in form of feature weights. At runtime, a hardware inference engine on the SM composes the feature vector, and uses the learned mapping to dynamically predict good warp scheduling decisions. To offset any statistical errors in prediction, the inference engine performs a neighbourhood search in the vicinity of the prediction. We evaluate *Poise* on disjoint set of benchmarks that were unseen during training, and observe a performance improvement of up to 2.94× and a harmonic mean speedup of 46.6% over the baseline GTO warp scheduler. *Poise* also outperforms the prior state-of-the-art warp scheduler by up to 141.1%, and an average of 15.1%.

In summary, we demonstrate a mechanism to achieve considerable accuracy and sophistication with a lightweight regression model. To arrive at a small, yet effective model, we apply domain knowledge through analytical reasoning, thereby considerably shrinking the feature vector to truly representative features. Therefore, our proposed technique drastically reduces the computational and storage needs of the learned model, making it suitable for architectural use. To further reduce the cost of adoption, we provide a software interface to change the feature weights, retaining the flexibility to retrain the model for newer applications. Through the above considerations, *Poise* demonstrates an effective way of applying machine learning to drive architectural decisions.

Chapter 6

Conclusion

In this thesis, we investigate the challenges arising due to the bandwidth bottlenecks present across the memory hierarchy in GPUs, with a particular focus on the cache hierarchy. We isolate three major factors that lead to the bandwidth bottlenecks: disproportionate bandwidth resources across the memory hierarchy; inefficient cache management policies; and high levels of thread-level parallelism. We show that the above factors lead to severe congestion across the memory hierarchy, leading to a breach in the latency tolerance property of the GPUs. This is because the high memory latencies arising due to congestion can no longer be hidden by multithreading and start to appear in the critical path of system performance.

To address and mitigate the bandwidth bottlenecks, we propose a three-pronged approach: cost-effective *scaling* of the existing bandwidth resources guided by an extensive characterization of the bandwidth bottlenecks; *supplementing* the existing bandwidth resources through an L1 cooperative caching network which exploits the presence of inter-core reuse in GPUs; and better *utilizing* the existing bandwidth resources by regulating the levels of multithreading and reducing cache thrashing. In Section 6.1, we further elaborate on the contributions made in this thesis, followed by a critical analysis of our proposals in Section 6.2. In Section 6.3, we conclude the chapter with a discussion on future research directions.

6.1 Contributions

In this section, we summarize and discuss the contributions made in the preceding chapters of this thesis.

6.1.1 Scaling the Bandwidth Resources

In Chapter 3, we show that bandwidth in the cache hierarchy is increasingly burdened due to the SMs injecting large volumes of memory requests on one side, and the high bandwidth off-chip memory injecting large volumes of memory responses on the other side. We further observe that the existing cache hierarchy is under-provisioned with respect to bandwidth resources to handle such high levels of memory traffic, resulting in significant levels of congestion. Therefore, we characterize the presence the bandwidth bottlenecks across the memory hierarchy in GPUs, including the cache hierarchy. This marks a departure from the traditional treatment of the bandwidth bottleneck where it is known to reside primarily in the off-chip memory. We quantify the stalls throughout the memory hierarchy and identify the architectural parameters that play a critical role in leading to a congested memory system. From the design space revealed by the characterization, we propose cost-effective configurations of the memory hierarchy to alleviate the bandwidth bottlenecks. We show that the performance improvement achieved by addressing the bandwidth bottleneck in the cache hierarchy often exceeds the speedup obtained by a memory system with a baseline cache hierarchy and HBM DRAM. We also show that scaling the bandwidth resources in isolation at specific levels of the memory hierarchy can be sub-optimal and can even be counter-productive. Therefore, we emphasize the need to resolve the bandwidth bottlenecks synergistically across different levels of the memory hierarchy.

6.1.2 Supplementing the Bandwidth Resources

In Chapter 4, we identify significant inter-core reuse in GPUs for general-purpose applications. The existing cache management policy aggravates the bandwidth bottleneck by sending duplicate memory requests for shared data that is already cached elsewhere at the same level in the memory hierarchy. Therefore, we use this opportunity to reuse data among L1 caches and reduce the bandwidth demand on the shared L2 cache. Furthermore, we show that there is a considerable leeway of around 80 cycles to fetch the shared data from a remote L1 cache, obviating the need for an aggressive or latency-sensitive scheme to share data. Therefore, we propose a Cooperative Caching Network (CCN) where we connect the L1 caches with a lightweight ring network to facilitate inter-core communication of shared data. The ring topology is lowest in terms of logical complexity and power consumption as all core-to-core connections are near-neighbour, and therefore, the wires are short. In addition, CCN scales linearly with

the increasing number of nodes, in contrast to the existing crossbar between L1 and L2 which scales polynomially. Therefore, the proposed cooperative caching network provides a cost-effective way to supplement the existing bandwidth between the L1 and L2 cache levels by exploiting the reuse characteristics of general-purpose applications.

6.1.3 Utilizing the Bandwidth Resources

In Chapter 5, we note that high levels of multithreading are normally desired in upcoming application domains. However, in memory-intensive applications, it causes severe contention for cache resources leading to cache thrashing, thereby aggravating the bandwidth bottleneck due to high cache miss rates. Therefore, we address the challenge of maintaining satisfactory levels of memory system performance without significantly sacrificing thread-level parallelism, in contrast to pure throttling schemes which severely limits parallelism. To this end, we propose a combination of machine learning and architecture techniques to dynamically determine the best warp scheduling decisions. Through these decisions, the warp scheduler alters the degree of thread-level parallelism in the system, and independently alters the number of warps appropriate to maintain satisfactory level of cache performance. The proposed mechanism, *Poise*, has two major components: a machine learning framework and a hardware inference engine. In the machine learning framework, we first use an analytical model to reveal the architecture and application features that influence the best warp scheduling decisions. Subsequently, we use a supervised learning model — trained offline on a set of profiled kernels — to learn a mapping from the set of extracted features to the best warp scheduling decisions. At runtime, the hardware inference engine collects the architecture and application features that were used during training, and uses the learned mapping to predict good warp scheduling decisions on previously unseen workloads. Therefore, *Poise* dynamically regulates the degree of thread-level parallelism in the GPU, and also independently improves the L1 caching efficiency by mitigating contention for cache resources. As a result, it reduces the high levels of memory requests sent to the lower levels of the memory hierarchy, thereby mitigating congestion.

6.2 Critical Analysis

In this section, we perform a critical analysis of the proposals presented in the prior chapters.

6.2.1 Ease of Adoption

The proposed mechanisms pose different thresholds for adoption in the future GPU architectures. Some of the metrics that determine such a threshold are hardware overheads, scalability and verification effort. We briefly evaluate our proposed schemes using the above metrics and argue about the comparative ease of adoption in the future architectures. The analysis is summarized in Table 6.1.

Hardware Overheads: Among the proposed mechanisms, cost-effective scaling and Cooperative Caching Network pose an area overhead of 1.5% and 1.3% respectively. On the other hand, *Poise* poses a minimal area overhead of less than 0.1%, as it only includes performance counters and finite-state machines. However, it is noteworthy that *Poise* does require one-off offline training of the regression model on a set of representative workloads, and minor software or compiler changes to communicate the learned feature weights to the hardware inference engine.

Scalability: The proposed cost-effective scaling uses an asymmetric crossbar, among other changes, which limits scalability due to the polynomial increase in cost incurred by the crossbar with increasing number of SMs. In contrast, the ring network proposed in Cooperative Caching Network scales linearly with respect to area and energy, and therefore it is a lucrative architectural choice for supplementing the bandwidth resources in the memory hierarchy. In CCN, however, the latency overhead of traversing the rings increases significantly with larger rings. Therefore, hierarchical implementation of the proposed ring networks can be used, thereby decomposing the serial latency of traversing the high number of nodes into concurrent transactions to multiple hierarchical rings. Finally, *Poise* scales linearly with respect to area and energy with increasing number of SMs. Moreover, due to independent warp schedulers on each SM, *Poise* does not pose any additional latency constraint in larger systems (unlike CCN), and therefore scales best among the proposed solutions. However, as the underlying architecture changes, *Poise* might require re-training of the regression model to account for a possible shift in the balance between thread-level parallelism and memory system performance arising out of architectural scaling.

Verification Effort: The proposed cost-effective scaling poses the least verification cost. This is because we restrict our design space exploration to *existing* architectural parameters rather than introducing new architectural components. As the existing configuration of the memory hierarchy is already verified, it would require minimal effort to verify the scaled configurations of the memory hierarchy. On the other hand, the

Table 6.1: Relative overhead of adoption for the proposed schemes indicated by H (highest), M (moderate) and L (lowest).

Metric	*Cost-effective scaling*	*Cooperative Caching Network*	*Poise*
Hardware Overhead	H	M	L
Scaling Overhead	H	M	L
Verification Effort	L	H	M

Cooperative Caching Network poses a significant verification cost due to an additional interconnection network. Some of the new scenarios to be verified are network deadlocks, protocol correctness, and timing correctness for the ring network and shadow tag arrays. Finally, *Poise* poses an intermediate and nominal verification effort as the majority of the hardware changes are restricted to the hardware warp scheduler.

6.2.2 Portability

In this thesis, we restrict our evaluation to NVIDIA GPUs as they are the most prevalent discrete GPUs, arguably holding the highest market share in the said domain. Consequently, we also restrict ourselves to CUDA programming model that is proprietary to NVIDIA platforms. However, we expect that the bandwidth bottlenecks are a pervasive problem present across different GPU architectures. This is indicated by the apparent success of high bandwidth memories such as GDDR5 and HBMs across architectures from different vendors. In fact, the recent introduction of High Bandwidth Cache Controller in AMD's Vega Architecture [7] indicates that the bandwidth bottleneck across the cache hierarchy is also attracting attention, and is not an isolated problem limited to NVIDIA GPUs. In summary, this thesis proves to be an effective primer to evaluate and mitigate the bottlenecks, subject to modification of the proposals based on the constraints of the underlying architecture.

6.2.3 Evaluation Methodology

In this thesis, we use a widely adopted cycle-accurate simulation based evaluation methodology — a general practice in architecture research. This is because of the exponential cost and time required to evaluate the architectural modifications in real chips, rendering it impractical. In addition, lack of standardized and open-source Register Transfer Level (RTL) IP modules for GPUs presents another limitation. Therefore, we

restrict ourselves to simulation infrastructures, trading off accuracy and cost for speed and ease of evaluation. However, it is noteworthy that we do bolster our proposals with sufficient intuitive reasoning and analytical modelling to ensure good reproducibility of results on real GPU chips.

Additionally, we simulate GPU architectures similar to Fermi (used in Chapter 3 and Chapter 4) and Kepler (used in Chapter 5) architectures. The more recent architectures, such as Maxwell and Pascal, are not currently supported due to the limitation of the existing simulation infrastructure available for academic research in the domain of GPUs. However, as the basic organization of the memory hierarchy is fairly consistent across different architectures, we expect our observations to be applicable to newer generations of GPUs as well.

6.2.4 Other Inefficiencies in the Memory Hierarchy

In this thesis, we address a few inefficiencies in the memory hierarchy pertaining to inadequate bandwidth resources in the cache hierarchy, poor cache management policies and high levels of multithreading. While we show that these inefficiencies are extremely critical, they are by no means an exhaustive list of factors that lead to bandwidth bottlenecks. There are several other limiting factors that can be addressed to improve the memory system performance. Some of these techniques include the usage of data compression [126, 88, 97] and changing the memory access granularity [130, 8] to reduce the bandwidth demand, DRAM scheduling mechanisms to improve off-chip bandwidth utilization [169, 87, 25, 26] and cooperative management of the shared bandwidth resources [37, 73].

6.3 Future Work

In this thesis, we focus on managing the GPU memory hierarchy with respect to bandwidth bottlenecks and aim to mitigate the congestion. As the number of SMs continue to grow and workloads continue to diversify, traditional organization of the memory hierarchy might become increasingly outdated. While we address some of the inefficiencies in the existing memory hierarchy, rethinking other assumptions and design decisions from traditional multiprocessors that may be suboptimal for GPUs is crucial and a potential research direction. One such possible area of research is in the domain of interconnection networks. With increasing number of nodes and considerably

higher request rates, traditional crossbar networks become highly saturated and lead to significantly higher queuing latencies, in addition to the polynomial growth in area and power. While recent research has shown promising results with low cost ring-like architectures in CPUs and GPUs [120, 95, 81, 172], it still remains an open problem for future research.

Another possible area of research is to explore the joint design space of multiple techniques that address different inefficiencies in GPUs. As future applications are expected to present an even higher demand on shared resources across the memory hierarchy, a collective redressal of the problem from multiple fronts is essential. For instance, in this thesis we explore efficient ways to scale, supplement and utilize the existing bandwidth resources in GPUs. Collectively, they form a large design space which can be tailored to synergistically address the bandwidth bottlenecks. Therefore, it is an important area of research to investigate the best ways to combine such techniques and harness the benefits of different proposals in a cost-effective manner.

In Chapter 4, we address the bandwidth implication of repeated accesses to the L2 cache for data shared across different cores. Such a policy not only leads to duplicate accesses to the L2 cache, but also leads to significant replication of data across private L1 caches. Storing duplicate data across different L1 caches reduces the effective on-chip storage provided by the already scarce L1 caches. A future research direction, therefore, is to explore sophisticated schemes that build upon the Cooperative Caching Network to reduce the redundancy in the on-chip storage. Such a scheme can improve the utilization of on-chip storage by selectively duplicating remote L1 data in a local L1 cache only when it exhibits high intra-core reuse, and prohibiting replication of remote L1 data if it is expected to show low intra-core reuse.

In Chapter 5, we focus on mitigating cache thrashing in the L1 cache, as it is one of the most scarce on-chip memory resource in GPUs. However, with growing workload sizes, lower levels of the memory — such as the shared L2 cache — will begin to suffer from cache thrashing as well, appearing as a potential bottleneck. Therefore, cache management policies that consider L1 and L2 caches collectively as a critical resource, is a potential area of research. For instance, with L2 cache becoming a critical resource, future machine learning based techniques can focus on optimally distributing the working set across both L1 and L2 caches, in contrast to focusing only on L1. Therefore, finding the optimal composition of warps with respect to their abilities to pollute the different levels of the memory hierarchy is an interesting research direction.

GPUs are increasingly used in datacenters and cloud computing, where different

SMs on a GPU may be executing different kernels from separate applications. In such a scenario, different SMs may pose varying need for shared resources. For instance, the shared memory resources — such as the on-chip and off-chip memory bandwidth — would appear more performance critical to SMs with few or no active warps, in contrast to SMs with sufficient active warps. Therefore, when such disparate SMs are present during the same epoch of execution, the criticality of the available memory bandwidth varies for different SMs. In the existing policy, however, a pending memory request tries to acquire a fraction of the shared memory bandwidth as soon as it is conceived, while being oblivious to the relative criticality of the shared bandwidth across other SMs. This is because each SM employs a greedy policy to acquire the shared bandwidth in order to maximize its own bandwidth utilization, thereby excessively depleting the shared bandwidth and causing congestion. It is analogous to the *Tragedy of the Commons*, a problem in economics where a strategy best for an individual in using an unregulated common resource may not yield the most optimal outcome for the group. In the recent years, microeconomics concepts such as Game Theory have found increasing application to solve similar problems in systems research [170, 106, 107, 171]. In GPUs, the problem of managing the shared system resources such as bandwidth and cache capacity falls under the same ambit of shared resource management. Therefore, a promising future direction is to further explore the use of microeconomic concepts to solve the above problems in GPUs.

6.4 Concluding Remarks

In this thesis, we draw attention to some of the inefficiencies in GPUs that lead to poor memory system performance. In the future, with increasing use of GPUs across diverse application domains and continual growth in workload complexity, the burden on the memory system is expected to rise even further. We hope that our work helps in understanding and addressing some of these problems, and motivates further research in this direction.

Bibliography

[1] M. Abdel-Majeed and M. Annavaram. Warped Register File: A Power Efficient Register File for GPGPUs. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture*, HPCA '13, pages 412–423, Washington, DC, USA, 2013. IEEE Computer Society.

[2] M. E. Acacio, J. González, J. M. García, and J. Duato. Owner Prediction for Accelerating Cache-to-cache Transfer Misses in a cc-NUMA Architecture. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, SC '02, pages 1–12, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[3] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using Machine Learning to Focus Iterative Optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 295–305, Washington, DC, USA, 2006. IEEE Computer Society.

[4] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson. GPU Concurrency: Weak Behaviours and Programming Assumptions. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 577–591, New York, NY, USA, 2015. ACM.

[5] J. Alsop, M. D. Sinclair, R. Komuravelli, and S. V. Adve. GSI: A GPU Stall Inspector to Characterize the Source of Memory Stalls for Tightly Coupled GPUs. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2016*.

[6] J. Alsop, M. S. Orr, B. M. Beckmann, and D. A. Wood. Lazy Release Consistency for GPUs. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–14, Oct 2016.

[7] AMD Radeon: Vega Architecture. `https://radeon.com/en-us/vega-architecture/`, 2017.

[8] A. Arunkumar, S. Y. Lee, and C. J. Wu. ID-cache: Instruction and Memory Divergence Based Cache Management for GPUs. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10, Sept 2016.

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

[10] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *ISPASS*, pages 163–174. IEEE, 2009.

[11] A. Bakhoda, J. Kim, and T. M. Aamodt. Throughput-Effective On-Chip Networks for Manycore Accelerators. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 421–432, Washington, DC, USA, 2010. IEEE Computer Society.

[12] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A Scalable Architecture Based on Single-chip Multiprocessing. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 282–293, June 2000.

[13] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs. In *Proceedings of the 22nd Annual International Conference on Supercomputing*, ICS '08, pages 225–234, New York, NY, USA, 2008. ACM.

[14] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA Code Generation for Affine Programs. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, pages 244–263, Berlin, Heidelberg, 2010. Springer-Verlag.

[15] B. M. Beckmann and D. A. Wood. Managing Wire Delay in Large Chip-Multiprocessor Caches. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 319–330, Washington, DC, USA, 2004. IEEE Computer Society.

[16] M. Bennasar, Y. Hicks, and R. Setchi. Feature Selection Using Joint Mutual Information Maximisation. *Expert Syst. Appl.*, 42(22):8520–8532, December 2015.

[17] C. M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, Inc., New York, NY, USA, 1995.

[18] R. Bitirgen, E. İpek, and J. F. Martinez. Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 318–329, Washington, DC, USA, 2008. IEEE Computer Society.

[19] P. Bodik, R. Griffith, C. Sutton, A. Fox, M. I. Jordan, and D. A. Patterson. Statistical Machine Learning Makes Automatic Control Practical for Internet Datacenters. In *Workshop on Hot Topics in Cloud Computing (HotCloud 09)*, 2009.

[20] S. Campanoni, K. Brownell, S. Kanev, T. M. Jones, G.-Y. Wei, and D. Brooks. HELIX-RC: An Architecture-compiler Co-design for Automatic Parallelization of Irregular Programs. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 217–228, Piscataway, NJ, USA, 2014. IEEE Press.

[21] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. P. O'Boyle, and O. Temam. Rapidly Selecting Good Compiler Optimizations Using Performance Counters. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 185–197, Washington, DC, USA, 2007. IEEE Computer Society.

[22] G. Chandrashekar and F. Sahin. A Survey on Feature Selection Methods. *Comput. Electr. Eng.*, 40(1):16–28, January 2014.

[23] J. Chang and G. S. Sohi. Cooperative Caching for Chip Multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, pages 264–276, Washington, DC, USA, 2006. IEEE Computer Society.

[24] J. Chang and G. S. Sohi. Cooperative Cache Partitioning for Chip Multiprocessors. In *Proceedings of the 21st Annual International Conference on Supercomputing*, ICS '07, pages 242–252, New York, NY, USA, 2007. ACM.

[25] N. Chatterjee, M. O'Connor, G. H. Loh, N. Jayasena, and R. Balasubramonian. Managing DRAM Latency Divergence in Irregular GPGPU Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 128–139, Piscataway, NJ, USA, 2014. IEEE Press.

[26] N. Chatterjee, M. O'Connor, D. Lee, D. R. Johnson, S. W. Keckler, M. Rhu, and W. J. Dally. Architecting an Energy-Efficient DRAM System for GPUs. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 73–84, Feb 2017.

[27] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.

[28] X. Chen, L. W. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W. M. Hwu. Adaptive Cache Management for Energy-Efficient GPU Computing. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 343–355, Dec 2014.

[29] R. L. Cheng. Video Game Console. US Design Patent 271311S, Dec 1983.

[30] H. Choi, J. Ahn, and W. Sung. Reducing Off-chip Memory Traffic by Selective Cache Management Scheme in GPGPUs. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 110–119, New York, NY, USA, 2012. ACM.

[31] G. Chrysos. Intel Xeon Phi Coprocessor - The Architecture. Technical report, Intel Corporation, 2012.

[32] CUDA C Programming Guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`, 2016.

[33] J. W. Davidson and S. Jinturkar. Memory Access Coalescing: A Technique for Eliminating Redundant Memory Accesses. In *Proceedings of the ACM SIGPLAN*

1994 Conference on Programming Language Design and Implementation, PLDI '94, pages 186–195, New York, NY, USA, 1994. ACM.

[34] S. Dublish, V. Nagarajan, and N. Topham. Characterizing Memory Bottlenecks in GPGPU Workloads. In *Proceedings of the 2016 IEEE International Symposium on Workload Characterization*, IISWC '16, Providence, Rhode Island, USA, 2016.

[35] S. Dublish, V. Nagarajan, and N. Topham. Cooperative Caching for GPUs. *ACM Trans. Archit. Code Optim.*, 13(4), December 2016.

[36] S. Dublish, V. Nagarajan, and N. Topham. Evaluating and Mitigating Bandwidth Bottlenecks Across the Memory Hierarchy in GPUs. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS, Santa Rosa, CA, USA, April 2017.

[37] S. Dublish. Student Research Poster: Slack-Aware Shared Bandwidth Management in GPUs. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, pages 451–452, New York, NY, USA, 2016. ACM.

[38] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum. Improving Cache Management Policies Using Dynamic Reuse Distances. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 389–400, Washington, DC, USA, 2012. IEEE Computer Society.

[39] A. ElTantawy, J. W. Ma, M. O'Connor, and T. M. Aamodt. A Scalable Multi-path Microarchitecture for Efficient GPU Control Flow. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 248–259, Feb 2014.

[40] CUDA by example - Errata, June 2014, 2014.

[41] W. C. Feng and S. Xiao. To GPU Synchronize or Not GPU Synchronize? In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pages 3801–3804, May 2010.

[42] GeForce 256: The World's First GPU. `http://www.nvidia.co.uk/page/geforce256.html`, 1999.

[43] J. Fox. *Applied Regression Analysis and Generalized Linear Models*. SAGE Publications, 2008.

[44] J. Friedman, T. Hastie, and R. Tibshirani. Regularization Paths for Generalized Linear Models via Coordinate Descent. *Journal of Statistical Software*, 33(1):1–22, 2010.

[45] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 407–420, Washington, DC, USA, 2007. IEEE Computer Society.

[46] G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, E. Bonilla, J. Thomson, H. Leather, C. Williams, M. O'Boyle, P. Barnard, E. Ashton, E. Courtois, and F. Bodin. MILEPOST GCC: Machine Learning based Research Compiler. In *GCC Summit*, Ottawa, Canada, June 2008.

[47] J. Gaur, M. Chaudhuri, and S. Subramoney. Bypass and Insertion Algorithms for Exclusive Last-level Caches. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 81–92, New York, NY, USA, 2011. ACM.

[48] Intel 82720: Graphics Display Controller. Datasheet, 1983.

[49] P. Glaskowsky. NVIDIA's Fermi: The First Complete GPU Computing Architecture. Technical report, NVIDIA, 2009.

[50] C. Gou and G. N. Gaydadjiev. Elastic Pipeline: Addressing GPU On-chip Shared Memory Bank Conflicts. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 3:1–3:11, New York, NY, USA, 2011. ACM.

[51] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Auto-tuning a High-level Language Targeted to GPU Codes. In *Innovative Parallel Computing*, pages 1–10, May 2012.

[52] N. Gulur, M. Mehendale, R. Manikantan, and R. Govindarajan. ANATOMY: An Analytical Model of Memory System Performance. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '14, pages 505–517, New York, NY, USA, 2014. ACM.

[53] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. C. Weiser. Many-Core vs. Many-Thread Machines: Stay Away From the Valley. *Computer Architecture Letters*, 8(1):25–28, 2009.

[54] M. A. Hall. *Correlation-based Feature Selection for Machine Learning*. PhD thesis, The University of Waikato, 1999.

[55] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. G. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton. Haswell: The Fourth-Generation Intel Core Processor. *IEEE Micro*, 34(2):6–20, 2014.

[56] HYNIX HBM. `https://www.skhynix.com/`, 2017.

[57] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A MapReduce Framework on Graphics Processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 260–269, New York, NY, USA, 2008. ACM.

[58] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[59] E. Herrero, J. González, and R. Canal. Distributed Cooperative Caching. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 134–143, New York, NY, USA, 2008. ACM.

[60] J. Hestness, S. W. Keckler, and D. A. Wood. A Comparative Analysis of Microarchitecture Effects on CPU and GPU Memory System Behavior. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 150–160, Oct 2014.

[61] M. A. Holliday and M. Stumm. Performance Evaluation of Hierarchical Ring-Based Shared Memory Multiprocessors. *IEEE Trans. Computers*, 43(1):52–67, 1994.

[62] S. Hong and H. Kim. An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness. In *Proceedings of the 36th Annual*

International Symposium on Computer Architecture, ISCA '09, pages 152–163, New York, NY, USA, 2009. ACM.

[63] GPUs for Cloud Servers. `https://www.ibm.com/cloud/gpu`, 2017.

[64] E. İpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz. 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, pages 195–206, New York, NY, USA, 2006. ACM.

[65] E. İpek, O. Mutlu, J. F. Martínez, and R. Caruana. Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 39–50, Washington, DC, USA, 2008. IEEE Computer Society.

[66] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer. Adaptive Insertion Policies for Managing Shared Caches. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 208–219, New York, NY, USA, 2008. ACM.

[67] A. Jaleel, H. H. Najaf-abadi, S. Subramaniam, S. C. Steely, and J. Emer. CRUISE: Cache Replacement and Utility-aware Scheduling. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 249–260, New York, NY, USA, 2012. ACM.

[68] V. Jatala, J. Anantpur, and A. Karkare. Scratchpad Sharing in GPUs. *ACM Trans. Archit. Code Optim.*, 14(2):15:1–15:29, May 2017.

[69] W. Jia, K. A. Shaw, and M. Martonosi. Characterizing and Improving the Use of Demand-fetched Caches in GPUs. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 15–24, New York, NY, USA, 2012. ACM.

[70] W. Jia, K. A. Shaw, and M. Martonosi. MRPB: Memory Request Prioritization for Massively Parallel Processors. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 272–283, Feb 2014.

[71] D. A. Jiménez and C. Lin. Dynamic Branch Prediction with Perceptrons. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, HPCA '01, pages 197–, Washington, DC, USA, 2001. IEEE Computer Society.

[72] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. 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, pages 395–406, New York, NY, USA, 2013. ACM.

[73] A. Jog, O. Kayiran, A. Pattnaik, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. Exploiting Core Criticality for Enhanced GPU Performance. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science, Antibes Juan-Les-Pins, France, June 14-18, 2016*, pages 351–363, 2016.

[74] S. Kaxiras and G. Keramidas. SARC Coherence: Scaling Directory Cache Coherence in Performance and Power. *IEEE Micro*, 30(5):54–65, September 2010.

[75] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das. Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 157–166, Sept 2013.

[76] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the Future of Parallel Computing. *IEEE Micro*, 31(5):7–17, September 2011.

[77] M. M. Keshtegar, H. Falahati, and S. Hessabi. Cluster-based Approach for Improving Graphics Processing Unit Performance by Inter Streaming Multiprocessors Locality. *IET Computers & Digital Techniques*, March 2015.

[78] M. Khairy, M. Zahran, and A. G. Wassal. Efficient Utilization of GPGPU Cache Hierarchy. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*, GPGPU-8, pages 36–47, New York, NY, USA, 2015. ACM.

[79] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. ATLAS: A Scalable and High-performance Scheduling Algorithm for Multiple Memory Controllers. In *IEEE*

16th International Symposium on High-Performance Computer Architecture (HPCA), pages 1–12, Jan 2010.

[80] H. Kim, J. Kim, W. Seo, Y. Cho, and S. Ryu. Providing Cost-effective On-chip Network Bandwidth in GPGPUs. In *2012 IEEE 30th International Conference on Computer Design (ICCD)*, pages 407–412, Sept 2012.

[81] H. Kim, G. Kim, S. Maeng, H. Yeo, and J. Kim. Transportation-network-inspired Network-on-Chip. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 332–343, Feb 2014.

[82] K. Kim, S. Lee, M. K. Yoon, G. Koo, W. W. Ro, and M. Annavaram. Warped-preexecution: A GPU Pre-execution Approach for Improving Latency Hiding. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 163–175, March 2016.

[83] D. B. Kirk and W.-m. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[84] J. Kloosterman, J. Beaumont, M. Wollman, A. Sethia, R. Dreslinski, T. Mudge, and S. Mahlke. WarpPool: Sharing Requests with Inter-warp Coalescing for Throughput Processors. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 433–444, Dec 2015.

[85] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way Multithreaded Sparc Processor. *IEEE Micro*, 25(2):21–29, March 2005.

[86] G. Koo, Y. Oh, W. W. Ro, and M. Annavaram. Access Pattern-Aware Cache Management for Improving Data Utilization in GPU. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 307–319, New York, NY, USA, 2017. ACM.

[87] N. B. Lakshminarayana, J. Lee, H. Kim, and J. Shin. DRAM Scheduling Policy for GPGPU Architectures Based on a Potential Function. *IEEE Computer Architecture Letters*, 11(2):33–36, 2012.

[88] S. Lal, J. Lucas, and B. Juurlink. E^2MC: Entropy Encoding Based Memory Compression for GPUs. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1119–1128, May 2017.

[89] S. Lawrence, C. L. Giles, and A. C. Tsoi. Lessons in Neural Network Training: Overfitting May be Harder than Expected. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island.*, pages 540–545, 1997.

[90] G. Lebanon. *Riemannian Geometry and Statistical Machine Learning*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2005. AAI3159986.

[91] A. R. Lebeck and D. A. Wood. Dynamic Self-invalidation: Reducing Coherence Overhead in Shared-memory Multiprocessors. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ISCA '95, pages 48–59, New York, NY, USA, 1995. ACM.

[92] B. C. Lee and D. Brooks. Efficiency Trends and Limits from Comprehensive Microarchitectural Adaptivity. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 36–47, New York, NY, USA, 2008. ACM.

[93] J. Lee and H. Kim. TAP: A TLP-aware Cache Management Policy for a CPU-GPU Heterogeneous Architecture. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.

[94] S. Y. Lee and C. J. Wu. Ctrl-C: Instruction-Aware Control Loop Based Adaptive Cache Bypassing for GPUs. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 133–140, Oct 2016.

[95] J. Lee, S. Li, H. Kim, and S. Yalamanchili. Design Space Exploration of On-chip Ring Interconnection for a CPU-GPU Heterogeneous Architecture. *J. Parallel Distrib. Comput.*, 73(12):1525–1538, December 2013.

[96] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu. Improving GPGPU Resource Utilization Through Alternative Thread Block Scheduling. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 260–271, Feb 2014.

[97] D. Lee, M. O'Connor, and N. Chatterjee. Reducing Data Transfer Energy by Exploiting Similarity within a Data Transaction. In *2018 IEEE 24th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2018.

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

[99] C. Li, S. L. Song, H. Dai, A. Sidelnik, S. K. S. Hari, and H. Zhou. Locality-Driven Dynamic GPU Cache Bypassing. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 67–77, New York, NY, USA, 2015. ACM.

[100] D. Li, M. Rhu, D. R. Johnson, M. O'Connor, M. Erez, D. Burger, D. S. Fussell, and S. W. Redder. Priority-based Cache Allocation in Throughput Processors. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 89–100, Feb 2015.

[101] L. Li, A. B. Hayes, S. L. Song, and E. Z. Zhang. Tag-Split Cache for Efficient GPGPU Cache Utilization. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, pages 43:1–43:12, New York, NY, USA, 2016. ACM.

[102] A. Li, S. L. Song, W. Liu, X. Liu, A. Kumar, and H. Corporaal. Locality-Aware CTA Clustering for Modern GPUs. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 297–311, New York, NY, USA, 2017. ACM.

[103] S.-w. Liao, T.-H. Hung, D. Nguyen, C. Chou, C. Tu, and H. Zhou. Machine Learning-based Prefetch Optimization for Data Center Applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 56:1–56:10, New York, NY, USA, 2009. ACM.

[104] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, March 2008.

[105] Y. Liu, E. Z. Zhang, and X. Shen. A Cross-input Adaptive Framework for GPU Program Optimizations. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–10, May 2009.

[106] Q. Llull, S. Fan, S. M. Zahedi, and B. C. Lee. Cooper: Task Colocation with Cooperative Games. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 421–432, Feb 2017.

[107] Q. Llull. *Microeconomic Models for Managing Shared Datacenters*. PhD thesis, Duke University, 2017.

[108] Luke Durant, Olivier Giroux, Mark Harris and Nick Stam. Inside Volta: The Worlds Most Advanced Data Center GPU. `https://devblogs.nvidia.com/inside-volta/`, 2017.

[109] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why On-chip Cache Coherence is Here to Stay. *Commun. ACM*, 55(7):78–89, July 2012.

[110] A. K. Mishra, O. Mutlu, and C. R. Das. A Heterogeneous Multiple Network-on-chip Design: An Application-aware Approach. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–10, May 2013.

[111] S. Mittal. A Survey of Techniques for Architecting and Managing GPU Register File. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):16–28, Jan 2017.

[112] O. Mutlu and T. Moscibroda. 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, pages 63–74, Washington, DC, USA, 2008. IEEE Computer Society.

[113] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt. Improving GPU Performance via Large Warps and Two-level Warp Scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 308–317, New York, NY, USA, 2011. ACM.

[114] R. Nasre, M. Burtscher, and K. Pingali. Atomic-free Irregular Computations on GPUs. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 96–107, New York, NY, USA, 2013. ACM.

[115] NVIDIA's Next Generation CUDA Compute Architecture: Fermi. Technical report, NVIDIA, 2009.

[116] NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. Technical report, NVIDIA, 2012.

[117] C. M. O'Brien. Negative Binomial Regression, Second Edition by Joseph M. Hilbe. *International Statistical Review*, 79(3):483–484, 2011.

[118] CUDA Occupancy Calculator. `https://developer.download.nvidia.com/compute/.../CUDA_Occupancy_calculator.xls`.

[119] Y. Oh, K. Kim, M. K. Yoon, J. H. Park, Y. Park, W. W. Ro, and M. Annavaram. APRES: Improving Cache Efficiency by Exploiting Load Characteristics on GPUs. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*, pages 191–203, 2016.

[120] H. Oi and N. Ranganathan. A Comparative Study of Bidirectional Ring and Crossbar Interconnection Networks. In *in Proceedings of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98), 883–890, Las Vegas*, 1998.

[121] M. A. O'Neil and M. Burtscher. Microarchitectural Performance Characterization of Irregular GPU Kernels. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 130–139, Oct 2014.

[122] M. S. Orr, S. Che, A. Yilmazer, B. M. Beckmann, M. D. Hill, and D. A. Wood. Synchronization Using Remote-Scope Promotion. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 73–86, New York, NY, USA, 2015. ACM.

[123] E. Park, L. N. Pouche, J. Cavazos, A. Cohen, and P. Sadayappan. Predictive Modeling in a Polyhedral Optimization Space. In *International Symposium on Code Generation and Optimization (CGO 2011)*, pages 119–129, April 2011.

[124] Paulius Micikevicius. Performance Optimization: Programming Guidelines and GPU Architecture Reasons Behind Them. `http://on-demand.gputechconf.com/gtc/2013/presentations/S3466-Programming-Guidelines-GPU-Architecture.pdf`, 2013.

[125] Z. Pawlak. *Rough Sets: Theoretical Aspects of Reasoning About Data*. Kluwer Academic Publishers, Norwell, MA, USA, 1992.

[126] G. Pekhimenko, E. Bolotin, N. Vijaykumar, O. Mutlu, T. C. Mowry, and S. W. Keckler. A Case for Toggle-aware Compression for GPU Systems. In *2016 IEEE*

International Symposium on High Performance Computer Architecture (HPCA), pages 188–200, March 2016.

[127] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood. Heterogeneous System Coherence for Integrated CPU-GPU Systems. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 457–467, New York, NY, USA, 2013. ACM.

[128] Parallel Thread Execution ISA, Version 4.3. `http://docs.nvidia.com/cuda/parallel-thread-execution`, 2016.

[129] G. Ravindran and M. Stumm. A Performance Comparison of Hierarchical Ring- and Mesh- Connected Multiprocessor Networks. In *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture*, HPCA '97, pages 58–, Washington, DC, USA, 1997. IEEE Computer Society.

[130] M. Rhu, M. Sullivan, J. Leng, and M. Erez. A Locality-aware Memory Hierarchy for Energy-efficient GPU Architectures. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 86–98, New York, NY, USA, 2013. ACM.

[131] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory Access Scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 128–138, New York, NY, USA, 2000. ACM.

[132] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-Conscious Wavefront Scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 72–83, Washington, DC, USA, 2012. IEEE Computer Society.

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

[134] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. In *Proceedings of the 13th ACM SIGPLAN*

Symposium on Principles and Practice of Parallel Programming, PPoPP '08, pages 73–82, New York, NY, USA, 2008. ACM.

[135] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-m. W. Hwu. Program Optimization Space Pruning for a Multithreaded GPU. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, pages 195–204, New York, NY, USA, 2008. ACM.

[136] J. Sartori and R. Kumar. Branch and Data Herding: Reducing Control and Memory Divergence for Error-Tolerant GPU Applications. *IEEE Transactions on Multimedia*, 15(2):279–290, Feb 2013.

[137] S. Seabold and J. Perktold. Statsmodels: Econometric and Statistical Modeling with Python. In *9th Python in Science Conference*, 2010.

[138] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A Many-core x86 Architecture for Visual Computing. In *ACM SIGGRAPH 2008 Papers*, SIGGRAPH '08, pages 18:1–18:15, New York, NY, USA, 2008. ACM.

[139] A. Sethia, D. Jamshidi, and S. Mahlke. Mascar: Speeding Up GPU Warps by Reducing Memory Pitstops. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 174–185, Feb 2015.

[140] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc. A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 11–22, New York, NY, USA, 2012. ACM.

[141] GPGPU-Sim Manual. `http://gpgpu-sim.org/manual`, 2014.

[142] M. D. Sinclair, J. Alsop, and S. V. Adve. Efficient GPU Synchronization Without Scopes: Saying No to Complex Consistency Models. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 647–659, Dec 2015.

[143] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt. Cache Coherence for GPU Architectures. *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 0:578–590, 2013.

[144] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014.

[145] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta Optimization: Improving Compiler Heuristics with Machine Learning. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 77–90, New York, NY, USA, 2003. ACM.

[146] J. A. Stratton, C. Rodrigrues, I.-J. Sung, N. Obeid, L. Chang, G. Liu, and W.-M. W. Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, Urbana, March 2012.

[147] J. A. Stuart and J. D. Owens. Efficient Synchronization Primitives for GPUs. *CoRR*, abs/1110.4623, 2011.

[148] G. Sun, C. Hughes, C. Kim, J. Zhao, C. Xu, Y. Xie, and Y. K. Chen. Moguls: A Model to Explore the Memory Hierarchy for Bandwidth Improvements. In *38th Annual International Symposium on Computer Architecture (ISCA), 2011*, pages 377–388, June 2011.

[149] D. Tarjan and K. Skadron. The Sharing Tracker: Using Ideas from Cache Coherence Hardware to Reduce Off-Chip Memory Traffic with Non-Coherent Caches. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[150] D. Tarjan, J. Meng, and K. Skadron. Increasing Memory Miss Tolerance for SIMD Cores. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 22:1–22:11, New York, NY, USA, 2009. ACM.

[151] J. V. Tu. Advantages and Disadvantages of Using Artificial Neural Networks versus Logistic Regression for Predicting Medical Outcomes. *Journal of Clinical Epidemiology*, 49(11):1225 – 1231, 1996.

[152] S.-Z. Ueng, M. Lathara, S. S. Baghsorkhi, and W.-m. W. Hwu. CUDA-Lite: Reducing GPU Programming Complexity. In *Languages and Compilers for Parallel Computing*, pages 1–15, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[153] V. Volkov. *Understanding Latency Hiding on GPUs*. PhD thesis, EECS Department, University of California, Berkeley, Aug 2016.

[154] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili. Dynamic Thread Block Launch: A Lightweight Execution Mechanism to Support Irregular Applications on GPUs. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 528–540, New York, NY, USA, 2015. ACM.

[155] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili. LaPerm: Locality Aware Scheduler for Dynamic Parallelism on GPUs. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 583–595, Piscataway, NJ, USA, 2016. IEEE Press.

[156] T. E. Westberg, C. S. Rode, and B. G. Burns. Video Computer System including Multiple Graphics Controllers and Associated Method. US Patent 4862156A, Aug 1989.

[157] Wolfram Mathworld. Least Squares Fitting. `http://mathworld.wolfram.com/LeastSquaresFitting.html`, 2018.

[158] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou. GPGPU Performance and Power Estimation Using Machine Learning. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '15, Burlingame, CA, USA, 2015.

[159] P. Xiang, Y. Yang, and H. Zhou. Warp-level Divergence in GPUs: Characterization, Impact, and Mitigation. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 284–295, Feb 2014.

[160] S. Xiao and W. Chun Feng. Inter-block GPU Communication via Fast Barrier Synchronization. In *IPDPS*, pages 1–12. IEEE, 2010.

[161] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang. Coordinated Static and Dynamic Cache Bypassing for GPUs. In *21st IEEE International Symposium on High Performance Computer Architecture, HPCA 2015, Burlingame, CA, USA, February 7-11, 2015*, pages 76–88, 2015.

[162] Q. Xu, H. Jeon, and M. Annavaram. Graph Processing on GPUs: Where are the bottlenecks? In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2014.

[163] S. Yan, G. Long, and Y. Zhang. StreamScan: Fast Scan Algorithms for GPUs Without Global Barrier Synchronization. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 229–238, New York, NY, USA, 2013. ACM.

[164] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A GPGPU Compiler for Memory Optimization and Parallelism Management. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 86–97, New York, NY, USA, 2010. ACM.

[165] Y. Yang, P. Xiang, M. Mantor, N. Rubin, and H. Zhou. Shared Memory Multiplexing: A Novel Way to Improve GPGPU Throughput. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 283–292, New York, NY, USA, 2012. ACM.

[166] Y. Yang, P. Xiang, M. Mantor, and H. Zhou. CPU-assisted GPGPU on Fused CPU-GPU Architectures. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.

[167] A. Yazdanbakhsh, B. Thwaites, H. Esmaeilzadeh, G. Pekhimenko, O. Mutlu, and T. C. Mowry. Mitigating the Memory Bottleneck With Approximate Load Value Prediction. *IEEE Design Test*, 33(1):32–42, Feb 2016.

[168] A. Yilmazer and D. Kaeli. HQL: A Scalable Synchronization Mechanism for GPUs. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 475–486, May 2013.

[169] G. L. Yuan, A. Bakhoda, and T. M. Aamodt. Complexity Effective Memory Access Scheduling for Many-core Accelerator Architectures. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 34–44, New York, NY, USA, 2009. ACM.

[170] S. M. Zahedi and B. C. Lee. REF: Resource Elasticity Fairness with Sharing Incentives for Multiprocessors. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 145–160, New York, NY, USA, 2014. ACM.

[171] S. M. Zahedi, Q. Llull, and B. C. Lee. Amdahl's Law in the Datacenter Era: A Market for Fair Processor Allocation. In *2018 IEEE 24th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2018.

[172] X. Zhao, S. Ma, C. Li, L. Eeckhout, and Z. Wang. A Heterogeneous Low-cost and Low-latency Ring-Chain Network for GPGPUs. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 472–479, Oct 2016.

[173] X. Zhao, S. Ma, Y. Liu, L. Eeckhout, and Z. Wang. A Low-cost Conflict-free NoC for GPGPUs. In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, pages 34:1–34:6, New York, NY, USA, 2016. ACM.

[174] X. Zhao, Y. Liu, A. Adileh, and L. Eeckhout. LA-LLC: Inter-Core Locality-Aware Last-Level Cache to Exploit Many-to-Many Traffic in GPGPUs. *IEEE Computer Architecture Letters*, 16(1):42–45, Jan 2017.

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