EFFICIENT UTILIZATION OF GPGPU CACHE HIERARCHY

By

Mahmoud Khairy Abdelsadek Abdallah

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Computer Engineering

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2015

EFFICIENT UTILIZATION OF GPGPU CACHE HIERARCHY

By
Mahmoud Khairy Abdelsadek Abdallah

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Computer Engineering

Under the Supervision of

Assoc. Prof. Amr G. Wassal

Associate Professor
Computer Engineering
Faculty of Engineering, Cairo University

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2015

EFFICIENT UTILIZATION OF GPGPU CACHE HIERARCHY

By
Mahmoud Khairy Abdelsadek Abdallah

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Computer Engineering

Approved by the
Examining Committee

_____
Assoc. Prof. Amr Galal-Eldeen Wassal, Thesis Main Advisor

_____
Assoc. Prof. Hossam Ali Hassan Fahmy, Internal Examiner

_____
Prof. Dr. Mohamed Watheq Ali Kamel El-Kharashi, External Examiner
Professor at Faculty of Engineering, Ain Shams University

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2015

Engineer's Name:	Mahmoud Khairy Abdelsadek Abdallah
Date of Birth:	13/6/1989
Nationality:	Egyptian
E-mail:	makhairy@eng.cu.edu.eg
Phone:	+201111365878
Address:	24 abdelaziz talaat hrab st., Elagouza, Giza
Registration Date:	01 / 10 / 2011
Awarding Date:	…./…./2015
Degree:	Master of Science
Department:	Computer Engineering Department
Supervisors:	
Examiners:	
	Prof. Mohamed Watheq Ali Kamel ElKharashi (External examiner)
	Assoc. Prof. Hossam Ali Hassan Fahmy (Internal examiner)
	Assoc. Prof. Amr Galal-Eldeen Wassal (Thesis main advisor)
Title of Thesis:	
	Efficient Utilization of GPGPU Cache Hierarchy

Key Words: Cache Management; GPGPU, Warp Throttling; Conflict-avoiding; Cache Bypassing.

Summary :

Throughput processors, such as GPGPUs, rely on massive multithreading to hide long memory latency. However, the high number of active threads GPGPU executes concurrently leads to severe cache thrashing and conflict misses. In this work, we propose a low-cost thrashing-resistant conflict-avoiding streaming-aware GPGPU cache management scheme that efficiently utilizes the GPGPU cache resources and addresses all the problems associated with GPGPU caches. The proposed method employs three orthogonal techniques. First, it dynamically detects and bypasses streaming applications. Second, a Dynamic Warp Throttling via Cores Sampling (DWT-CS) is proposed to alleviate cache thrashing. DWT-CS runs an exhaustive searching over cores to find the best number of warps that achieves the highest performance. Third, we employ a better cache indexing function, Pseudo Random Interleaving Cache (PRIC), that is based on polynomial modulus mapping, to mitigate associativity stalls and eliminate conflict misses. Our proposed method improves the average performance of streaming and contention applications by 1.2X and 2.3X respectively.

Acknowledgements

Before anyone else, I would like to thank Allah, The most Gracious, The most Merciful, for inspiring me with the idea of this thesis, giving me the strength, patience, and power to accomplish this research work.

Then, it comes to my advisor, Dr. Amr Wassal, for his great efforts, guidance, suggestions, encouragement and prompt responses to my questions and inquiries.

Special thanks go to Dr. Mohamed Zahran for his contributions through technical advice and feedback which helped me to overcome hurdles and made this work a wonderful learning experience.

I would like to thank Mohamed Hammad for his insightful feedback on this thesis. We also thank Wenhao Jia, a PhD student at Princeton University, and Tim Rogers, a PhD student at University of British Columbia, for generously sharing the source code of Memory Request Prioritization Buffer and Cache-Conscious Wavefront Scheduler respectively. I should also thank Ahmed ElTantawy, a PhD student at University of British Columbia, for his assistance with GPGPU-sim simulator and Latex tools.

Finally, I like to dedicate this work to my parents, for their great support and prayers, and presenting all forms of support.

Table of Contents

List of Tables

List of Figures

Abstract

Modern GPUs are equipped with general-purpose L1 and L2 caches in an attempt to reduce memory bandwidth demand and improve the performance of some irregular GPGPU applications. However, due to the massive multithreading, GPGPU caches suffer from severe resource contention and low data-sharing which may degrade the performance instead.

In this work, we propose three techniques to efficiently utilize and improve the performance of GPGPU caches. The first technique aims to dynamically detect and bypass memory accesses that show streaming behavior. In the second technique, we propose dynamic warp throttling via cores sampling (DWT-CS) to alleviate cache thrashing by throttling the number of active warps per core. DWT-CS monitors the MPKI at L1, when it exceeds a specific threshold, all GPU cores are sampled with different number of active warps to find the optimal number of warps that mitigates thrashing and achieves the highest performance. Our proposed third technique addresses the problem of GPU cache associativity since many GPGPU applications suffer from severe associativity stalls and conflict misses. Prior work proposed cache bypassing on associativity stalls. In this work, instead of bypassing, we employ a better cache indexing function, Pseudo Random Interleaving Cache (PRIC), that is based on polynomial modulus mapping, in order to fairly and evenly distribute memory accesses over cache sets.

The proposed techniques improve the average performance of streaming and contention applications by 1.2X and 2.3X respectively. Compared to prior work, it achieves 1.7X and 1.5X performance improvement over Cache-Conscious Wavefront Scheduler and Memory Request Prioritization Buffer respectively.

Chapter 1: Introduction

1.1 Introduction

Throughput-oriented processors, such as General Purpose Graphics processing Units (GPGPUs), have been widely adopted for accelerating compute-intensive data-parallel applications due to their high computational power and energy efficiency [22, 23, 28, 75, 112, 117]. However, GPGPU programming is a difficult task. The programmer has to explicitly manage the on-chip scratchpad memory to generate coalesced memory accesses and exploit data locality [84, 121]. Further, it has been shown that the memory throughput has become a limiting factor for many GPGPU applications performance [75]. To address these issues, modern GPUs [114, 156] are equipped with general purpose on-chip cache hierarchy in an attempt to reduce off-chip memory bandwidth demand, increase memory system throughput, improve the performance of some irregular GPGPU applications and enhance the GPU programmability.

GPU cache size is very limited, compared to the number of active threads GPU executes concurrently. Table 1.1 compares the L1 cache capacity per thread for CPUs vs GPUs. As shown in table, recent NVIDIA's Fermi GPU [156] supports 1536 active threads per core, and L1 cache size is configurable to 16KB or 48KB. Thus, the average L1 cache capacity per thread is only 10 or 32 bytes, which is less than a single cache line size (=128 bytes). This behavior is also found in NVIDIA's Kepler GPU [114] that has 2048 active threads per core and a read-only L1 data cache of size 48KB. This means that the GPU cache is not designed to keep the per-thread working set, as it is the case in CPU (for example, Intel core i7 CPU [52] contains 2 threads per core, 32KB L1, thus 16KB per thread). In fact, GPU caches were designed to exploit some access patterns that exhibit small cache footprint per thread and can fit in the cache (e.g. spilled registers, small-stride access pattern [121] and inter-core data locality [114]). In case GPGPU applications rely on caches to exploit data locality and they contain a large cache footprint per-thread, the active threads will compete on the few available cache lines and the L1 cache will be susceptible to *thrashing*. Moreover, the limited number of set associativity, typically between 4-6 [106], makes the L1 cache more vulnerable to *associativity* stalls and *conflict* misses. In addition, many GPGPU applications use the scratchpad memory to exploit locality. These applications show a *streaming* behavior on L1 cache when they are cached. Cache management schemes that are unaware of these streaming applications cause useless unintended contention at L1 cache and this may hurt the performance instead. For these applications, it is better to bypass L1 and/or L2 cache.

CTA throttling [73, 95], Warp throttling [98, 132, 133, 171], FIFO buffer [69] and thrashing-resistant cache replacement policy [25, 26, 98] are different techniques which have been proposed to alleviate cache thrashing, while cache bypassing [69, 98, 171] was proposed to mitigate the associativity stalls. However, many of these proposals address the cache thrashing problem only, incur a considerable storage overhead and require significant changes to the baseline architecture. Further, the proposed cache bypassing to handle associativity stalls is not an effective method to efficiently utilize the available cache resources. In many cases, bypassing occurs while cache sets are underutilized.

Table 1.1: L1 cache capacity per thread for CPU vs GPU [98]

	Intel Corei7	IBM Power7	NVIDIA Fermi	NVIDIA Kepler
L1 cache size	32KB	32KB	16-48KB	48KB
Threads per Core	2	4	1536	2048
L1/thread	16KB/thread	8KB/thread	10-32B/thread	24B/thread

In this work, we propose a low-cost thrashing-resistant conflict-avoiding streaming-aware GPGPU cache management scheme that efficiently utilize the GPGPU cache resources. The proposed method employs three techniques. First, it dynamically detects and bypasses streaming applications that show streaming behavior in L1 or L2 cache. Second, we propose dynamic warp throttling via cores sampling (DWT-CS) to alleviate cache thrashing. DWT-CS monitors the MPKI at L1, when it exceeds a specific threshold, all GPU cores are sampled with different number of active warps. Then, the active warps per all cores will be throttled to the number of warps that is associated with the winner core (the core which achieved the highest performance during the sampling period). Third, we employ a better cache indexing function, Pseudo Random Interleaving Cache (PRIC), that is based on polynomial modulus mapping [127], to mitigate associativety stalls and eliminate conflict misses. PRIC near-randomly and fairly distributes memory accesses over cache sets and thus efficiently utilizes the cache resources.

1.2 Contribution

This thesis makes the following contributions:

1) We analyze and measure the amount of locality that exist in GPGPU workloads by using fully-associative unbounded caches. We show that many GPGPU applications have large working set or poor cache reuse and don't benefit from the cache hierarchy, while other applications exhibit a high level of cache thrashing and/or associativity contention.

2) We propose a low-cost thrashing-resistant conflict-avoiding streaming-aware cache management scheme that addresses all the problems associated with GPGPU caches.

3) Prior work proposed cache bypassing on associativity stalls. In this work, instead of bypassing, we employ a better cache indexing function, that is based on polynomial modulus mapping.

4) Compared to prior work, our method has simpler hardware and achieves a harmonic mean 1.7X and 1.5X performance improvement over Cache-Conscious Wavefront Scheduler (CCWS) and Memory Request Prioritization Buffer (MRPB) respectively.

This work resulted in the following publications:
Mahmoud Khairy, Mohamed Zahran, Amr G. Wassal. "Efficient Utilization of GPGPU Cache Hierarchy" at the Proceedings of 8th Workshop on General Purpose Computing using GPUs (GPGPU8), co-located with 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP), San Francisco, CA, February 2015

Mahmoud Khairy, Mohamed Zahran, Amr G. Wassal. "SACAT: Streaming-Aware Conflict-Avoiding Thrashing-Resistant GPGPU Cache Management Scheme", *in preparation 2015*.

Mahmoud Khairy, Amr G. Wassal " A Survey of Architectural Support for Improving GPGPU Performance, Programmability and Heterogeneity", *in preparation 2015*.

1.3 Organization of the thesis

The rest of this thesis is organized as follows, Chapter 2 describes our baseline architecture, literature survey is discussed in Chapter 3, Chapter 4 describes experimental setup and workload characterization, Chapter 5 describes our proposed techniques, Chapter 6 presents the experimental results, and Chapter 7 concludes.

Chapter 2: Background

In this chapter we give a short overview on GPGPU programming model and architecture. It is important to note that, in this chapter, we mainly describe the basics of GPGPUs in order to understand the rest of this thesis and grasp the contribution of this work. For more details on GPGPU programming model and architecture, we kinldy refer the reader to [84, 116].

2.1 GPGPU Programming Model

The CUDA [116] or OpenCL [77] programming model allows the programmers to express the data level parallelism in terms of fine-grain scalar threads. A typical GPGPU application consists of multiple kernels (or grids). Figure 2.1 (a) shows a scalar program (i.e. sequential code) that increments all the elements of an arbitrary 2D matrix. The scalar program is written in sequence to be executed on a sequential processor (i.e. traditional CPU). Figure 2.1 (b) shows the corresponding CUDA program that implements the same function. In this example, the CUDA application is composed of one kernel. However, a CUDA application can have multiple kernels. The CUDA program expresses the data-level parallelism found in the scalar code in terms of fine-grain threads. Then, these fine threads are executed on a massively data-parallel processor, like GPGPUs.

CUDA Thread Hierarchy A typical CUDA kernel (or grid) is composed of two-level thread hierarchy to help the programmer to map and distribute data workload among threads. The higher level of hierarchy is a 2-dimensional thread block (a.k.a. Cooperative Thread Array or CTA), and each thread block consists of 2-dimensional scalar threads [1]. For instance, Figure 2.2 depicts the CUDA two-level thread hierarchy corresponding to the CUDA program shown in Figure 2.1 (b). As shown in figure, the higher level of hierarchy is a group of 2x2 thread blocks. Each thread block is composed of 2-dimensional 2x4 scalar threads. Threads within the same thread block communicate with each other through a shared on-chip scratchpad memory and synchronization primitives. The programmer specifies the grid and thread block dimensions at the kernel launch (line 3 in CUDA program). The first two lines in the CUDA kernel (line 6 and 7) calculate global indices i and j of the element that the thread should increment. The two indices i and j are calculated using predefined CUDA-specific global thread block ID (X and Y) and local thread ID (X and Y) within the thread block that this thread belongs to.

There are two important aspects of the GPGPU programming model. First. It follows a CPU offloading model, which means that the programmer is responsible for moving the data from CPU memory to GPU memory (line 2 in CUDA program), and when the kernel execution is finished, he has to move the result from GPU memory back to CPU memory (line 4 in CUDA program). Second, the GPGPU programming model is a Single Program Multiple Data (SPMD) model, which means that all threads run the same program (i.e. kernel), however each thread can be at different states within the CUDA program. In other words, threads do not necessarily execute the same instruction at a time.

[1] Current GPUs support 3-dimensional threads hierarchy

```
//Scalar program

1. float A[4][8];

2. do-all(i=0;i<4;i++){
3.   do-all(j=0;j<8;j++){
4.       A[i][j]++;
   }
}
```

(a) Scalar Program

```
//CUDA program

1. float A_h[4][8];

2. cudaMemcpy(A_d, A_h, size, HtD);

3. kernelF<<<(2,2),(4,2)>>>(A_d);

4. cudaMemcpy(A_h, A_d, size, DtH);


5. __device__    kernelF(A_d){
6.       i = blockDim.x * blockIdx.y + blockIdx.x;
7.       j = threadDim.x * threadIdx.y + threadIdx.x;

8.      A[i][j]++;
}
```

(b) CUDA Program

Figure 2.1: Scalar Program vs CUDA Program

SIMT Model During run-time, each consecutive 32 threads are grouped together to formulate a warp (a.k.a. wavefront). Warps are executed in a Single Instruction Multiple-Threads (SIMT) model. In SIMT execution model: (1) all threads within the same warp execute the same Program Counter (PC), i.e. execute in a lock-step, and each warp has its own PC. This amortizes instruction fetch and decode cost improving efficiency. (2) Threads are allowed to follow different control flow paths. (3) A long memory latency is tolerated by a zero-overhead warp switching. Next section, we discuss the SIMT execution model in details.

2.2 GPGPU Architecture

Our baseline GPGPU, shown in Figure 2.3, consists of multiple GPU cores, named Streaming Multiprocessors (SMs), [2] and a group of memory partitions. Each SM has its own register file, private L1 data cache, read-only texture cache, constant cache and software-managed scratchpad memory, named shared memory. They also contain a group of execution units, such as single instruction multiple data units (SIMDs) and special function units (SFUs). Each memory partition has a slice of the L2 cache and a GDDR5 memory controller that are shared among the SMs. The SMs and the memory partitions are connected via an interconnection network (i.e. on-chip network).

[2] In this thesis, we use the terms GPU core and Streaming Multiprocessor interchangeably.

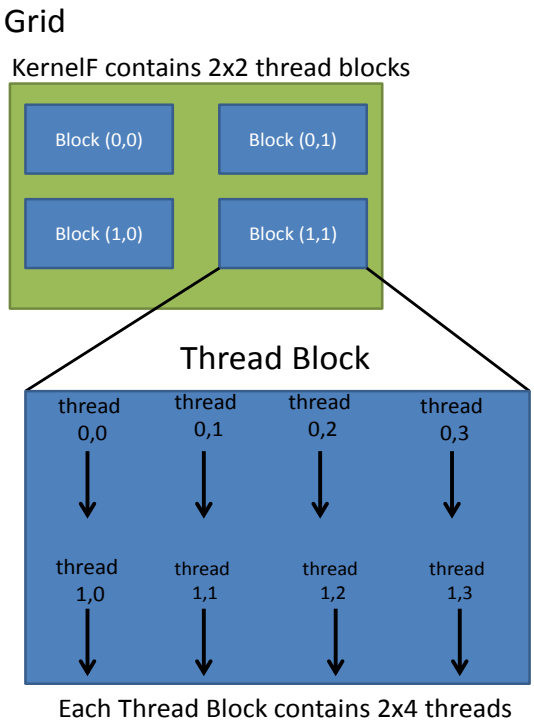

Figure 2.2: CUDA Thread Hierarchy

Warp Scheduling The warps in GPGPUs are scheduled in a two-level hierarchy (thread block scheduling and warp scheduling). A thread block scheduler, as shown in Figure 2.4, distributes the thread blocks among SMs in a load-balanced round-robin [13] fashion. Thread block is dispatched to a SM only if the required resources of the thread block are available on this SM (e.g. register file, shared memory, warp scheduler entries, etc.). Thread block are subdivided by hardware into warps (each warp is composed of a consecutive 32 threads). Each SM contains a number of warp schedulers. The warp scheduler employs a specific policy to schedule the available warps over the execution units. Figure 2.5 shows pictorially an example of round-robin warp scheduling example. As shown in figure, warp priority rotates every cycle. Greedy-then-oldest (GTO) is another scheduling policy that runs a single warp until it stalls then picks the oldest ready warp [132]. Each SM contains a memory-coalescing unit that attempts to coalesce memory requests of active threads within each warp into the fewest possible cache line-sized memory requests.

Control Flow Handling Recall that current GPGPU hardware implementations employ SIMT execution model in which only one instruction is fetched for all the threads within the same warp. However, a *control flow divergence* occurs when threads in the same warp execute different control flow paths due to branch/loop statement. In this case, GPGPU requires a mechanism to allow each thread to follow its own thread of control. Current GPGPUs handle control flow divergence and re-convergence with a hardware-based stack reconvergence [42]. In stack-based scheme, the execution of divergent paths is serialized and it ensures reconvergence occurs at or before the immediate postdominator

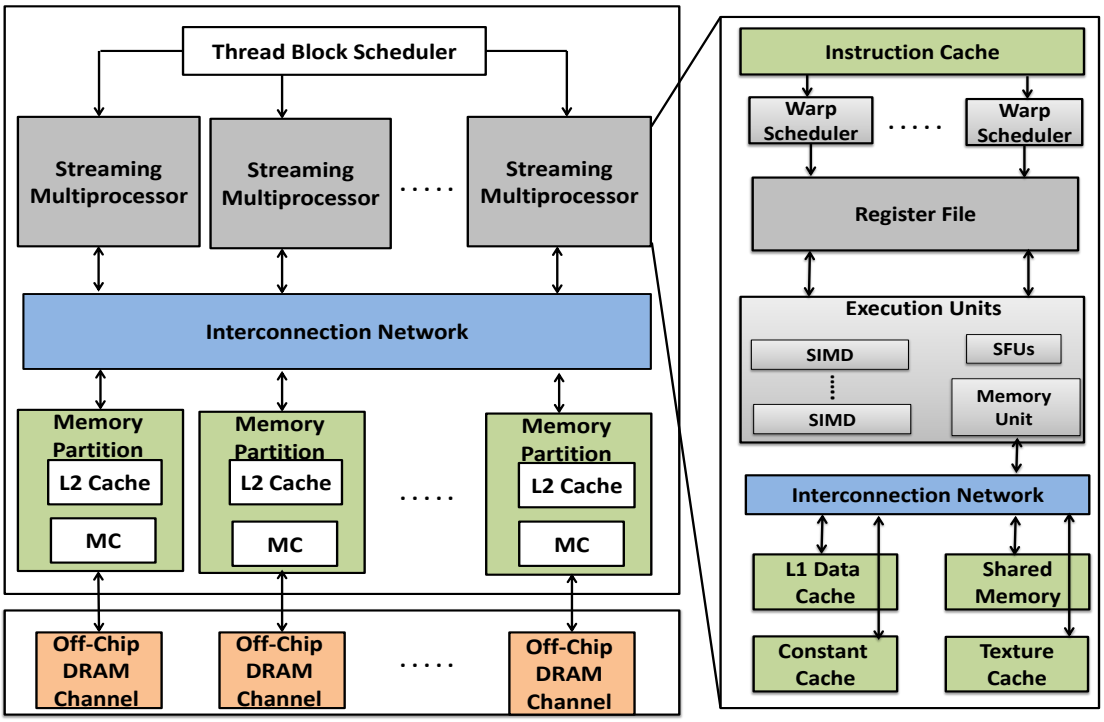

Figure 2.3: Baseline GPGPU Architecture

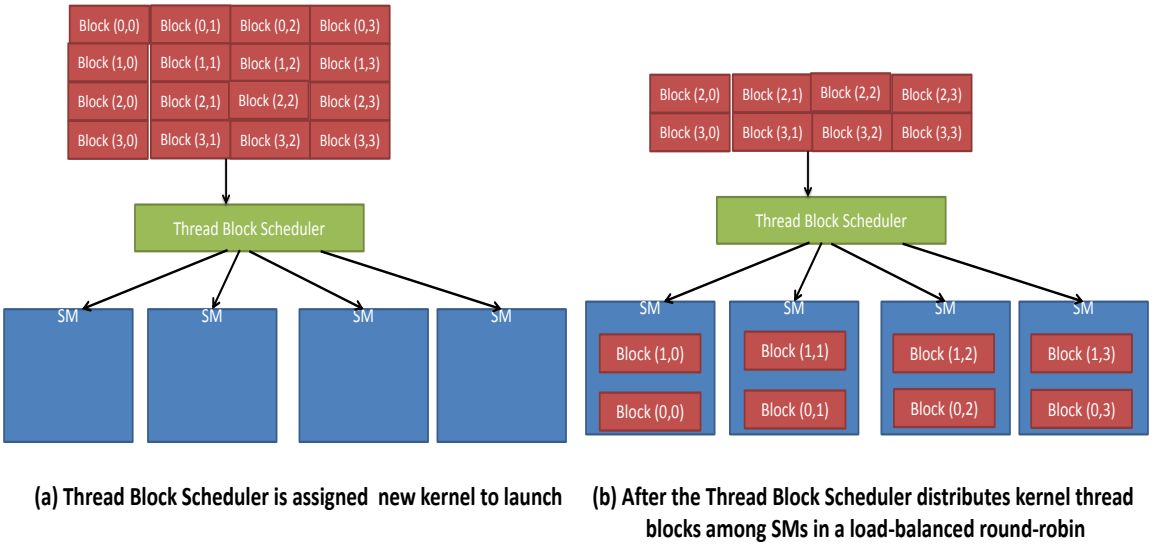

Figure 2.4: Load-balanced Round-Robin Thread Block Scheduling Example

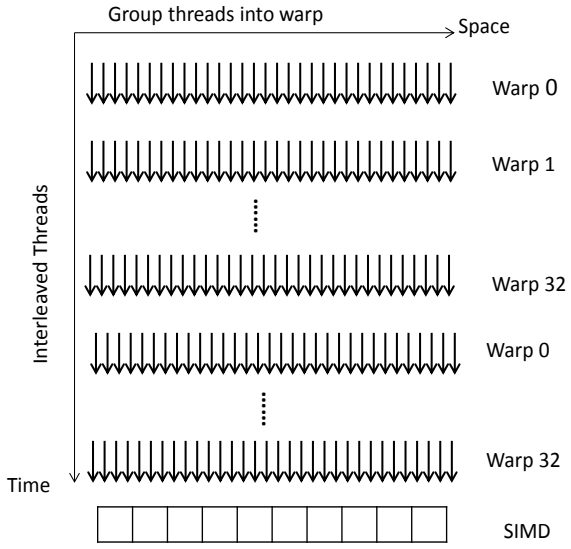

Figure 2.5: Round Robin Warp Scheduling

(IPDOM) of the divergent branch. Figure 2.6 depicts the reconvergence stack and its operation on the example control flow shown in Figure 2.6(a). As shown in figure, a divergence stack entry consists of three fields: a re-convergence PC, an active mask, and an execute PC. The hardware reconvergence stack tracks the program counter (PC) associated with each control flow path, which threads are active at each path (the active mask of the path), and at what PC should a path reconverge (RPC) with its predecessor in the control-flow graph. Since a warp can only have a single active PC at any given time, when branch divergence occurs, one path must be chosen first and the other is pushed on a divergence stack associated with the warp so that it can be executed later. Figures 2.6(b) through (e) show the state of a warps PC, active mask, and divergence stack at relevant points in time as it executes the control flow graph of Figure 2.6(a).

GPU vs CPU design philosophy (Throughput-oriented vs Latency-oriented) CPU and GPU are different architectures. Each has its own design philosophy. There is no one better than other, since each was designed to execute a specific task in an efficient manner. Table 2.1 summarizes the differences between both architectures. CPUs are well designed to execute sequential code. They are called Latency-oriented architecture, because they are optimized to tolerate long latency instruction and memory access. They devote more transistor area (as shown in figure 2.7) for sophisticated control (e.g. branch predication and data forwarding) to tolerate long latency instruction. They contain larger last level caches (up to 8 MB in current CPUs) in order to hide long memory latency. On the other hand, GPUs are throughput-oriented architecture. They are well designed to execute some parts of applications that contain data-level parallelism (e.g. for/while loops) which requires high arithmetic intensity and large memory throughput. As shown in figure 2.7, GPUs devote most of the chip area for processing elements (i.e ALUs) that are deeply pipelined to increase the arithmetic throughput. They employ a simpler logic control (no branch prediction) and small caches. GPGPUs rely on massive number of threads/warps interleaved with each other with zero-overhead switching in order to hide long memory

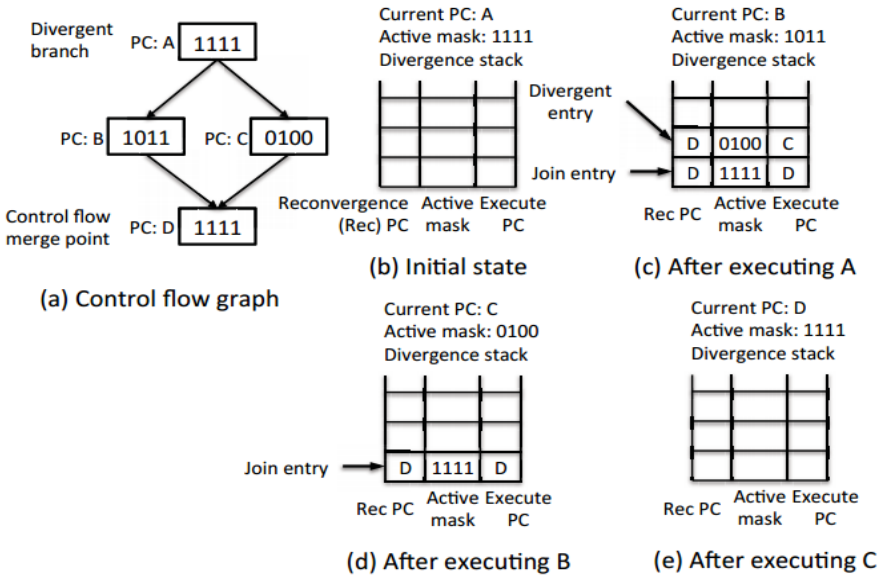

Figure 2.6: A stack-based divergent branch handling mechanism [22]

latency and instruction.

Figure 2.8 and 2.9 compares between CPUs and GPGPUs in terms of computational intensity (GFLOPs) and memory bandwidth (GB /sec) respectively. As shown in figures, GPUs achieve a high computational power and memory bandwidth compared to CPUs. The recent NVIDIA GTX 680 outperforms Intel Sandy Bridge in terms of GFLOPs and memory bandwidth by 6X and 4X respectively. However, as we stated earlier, this does not mean that GPUs are much more powerful and better than CPUs. GPUs are better when it comes to data-parallel high-computational code (e.g. loops). On the other hand, CPUs are better when it comes to sequential latency-sensitive code (e.g. heavily sequential branches). To take advantage of both CPUs and GPUs, and efficiently utilize the available hardware resources, programmers need to execute the serial parts of their code on CPUs and launch data-parallel parts on GPUs. By this way, they are able to achieve the most out of the underlining hardware. CPU-GPU programming is well known as heterogeneous computing [84, 116].

Table 2.1: CPU vs GPU comparison

	CPU	**GPGPU**
Architecture design philosophy	Latency-oriented architecture	Throughput-oriented architecture
Microarchitecture attributes	Large caches, Sophisticated control, Branch prediction , Data forwarding	Small caches, simple control, No branch prediction, No data forwarding
Threads per core	2-4 Threads per core	768-1024 threads (32-64 warps) per core
Hiding memory latency strategy	Hide memory latency through large Cache Hierarchy	Hide memory latency through massive multithreading zero-overhead switching
Preferable code execution	Sequential Code (if statement, function calling, ...)	Data-parallel Code (loops, streams, ...)

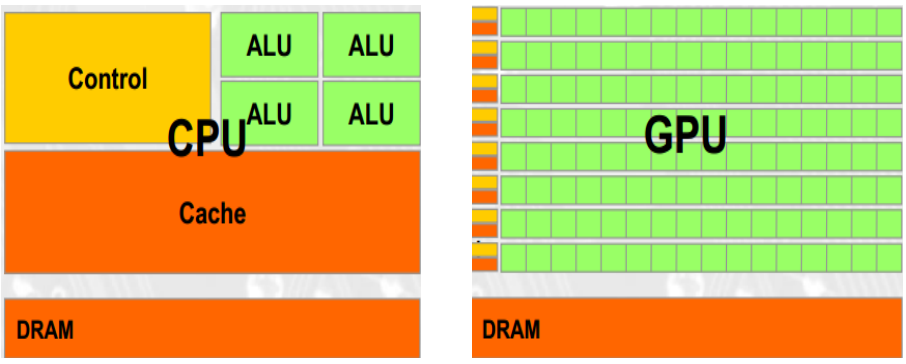

Figure 2.7: CPU vs GPU design philosophy [116]

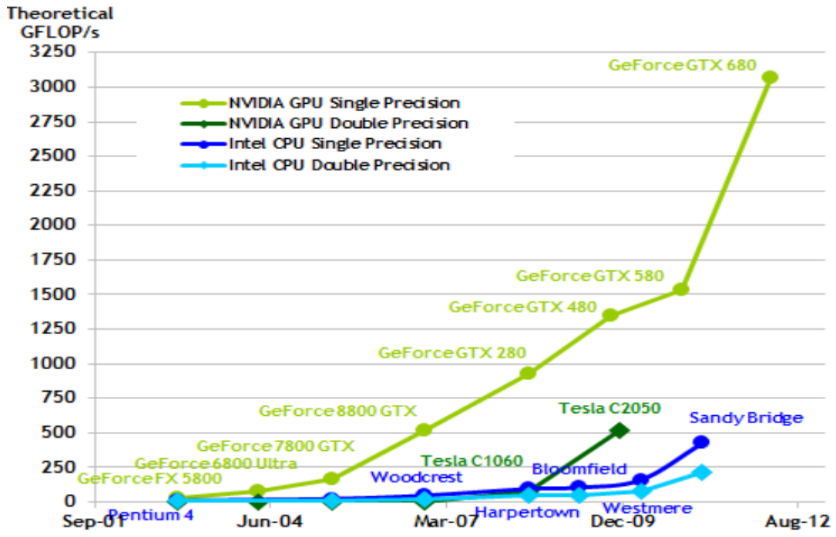

Figure 2.8: CPU vs GPU GFLOPS [116]

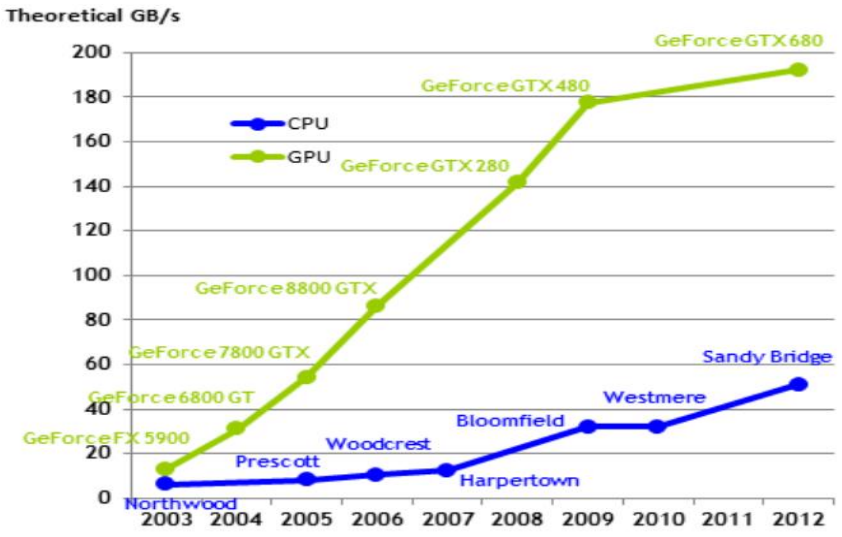

Figure 2.9: CPU vs GPU Memory Bandwidth [116]

Chapter 3: Literature Review

Recent years have seen a trend in using Graphics Processing Units (GPU) for General Purpose Computing. This trend was allowed by the high computational power and the efficient performance per watt the GPUs can achieve compared with multicore CPUs. Moreover, current heterogeneous chip multiprocessor integrate a GPU architecture on a die which increases the availability and utilization of GPUs as a general purpose data-parallel accelerator. However, the irregular execution flow of some workloads and the complexity of GPGPU programming, limits the range of applications that may benefit from GPUs. This chapter surveys research works on architectural support and software techniques to improve performance and programmability of GPUs for general purpose computing. It also provides a survey on research works which aim to enhance the on-chip integration of CPU-GPU heterogeneous architecture.

3.1 Introduction

Graphics Processing Units (GPUs) have been used for several years as a fixed-function hardware accelerator for 3D Graphics applications. Earlier generations of GPUs were designed to implement the conventional 3D rendering pipeline [78, 100]. However, the high computational power which the GPUs can achieve compared with multicore CPUs, encourage the developers to use GPUs for compute-intensive non-graphics workloads [149]. At this time, the term General Purpose computing using Graphics Processing Units (GPGPU) has emerged widely. The programmers used graphics APIs (e.g. Direct3D or OpenGL) to access shader cores. The programmers had to map program data appropriately to the available shader buffers and manage the data accurately through the graphics pipeline. Obviously, using graphics APIs for non-graphics general purpose programming was a very difficult task. However, with some heroic efforts, considerable speedups were achieved [118]. This trend prompted the GPU vendors to build a more programmable GPU architecture, known as unified shader architecture (e.g. NVIDIA's Tesla [101], NVIDIA's Fermi [112] and AMD Evegreen [39]) and release more-friendly high level abstraction APIs to facilitate the ease of GPGPU programming (e.g. NVIDIA's CUDA [116], AMD's CTM [60] and OpenCL [77]). Since then, a new era of GPGPU architecture and programming was unleashed and is still evolving to this day [22, 23, 28, 31, 44, 75, 112, 117].

Figure 3.1 depicts the number of research papers related to GPGPUs that were published in the last seven years at the top-tier computer architecture conferences. As shown in figure, there was a great interest in GPGPUs during the last two years. Up to 28 research papers were published in the last year (2014) and they represent almost 15% of the total number of papers. Figure 3.2 characterizes and divides these papers into different categories. As we can see, the researchers hardly worked on improving the performance of GPGPUs by alleviating on-chip resource contention and mitigating the impact of control flow divergence. Since GPGPU programming is very hard and complex, researchers also worked on enhancing the GPGPU programmability. They addressed this problem by equipping GPGPUs with some hardware support to improve data sharing and synchronization (e.g. cache coherence and consistency). They also investigated new

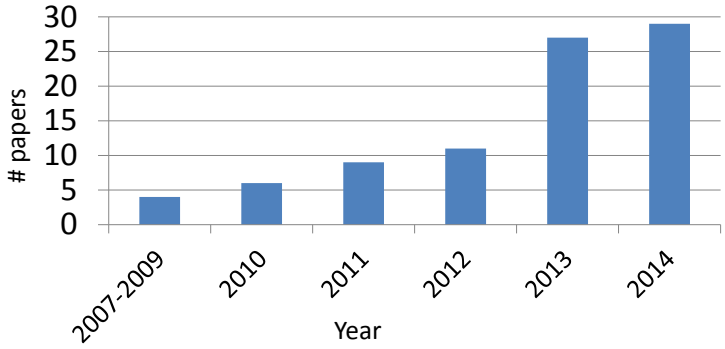

Figure 3.1: Number of research papers related to GPGPU published during the last seven years at the top-tier computer architecture conferences (MICRO, ISCA, ASPLOS, and HPCA)

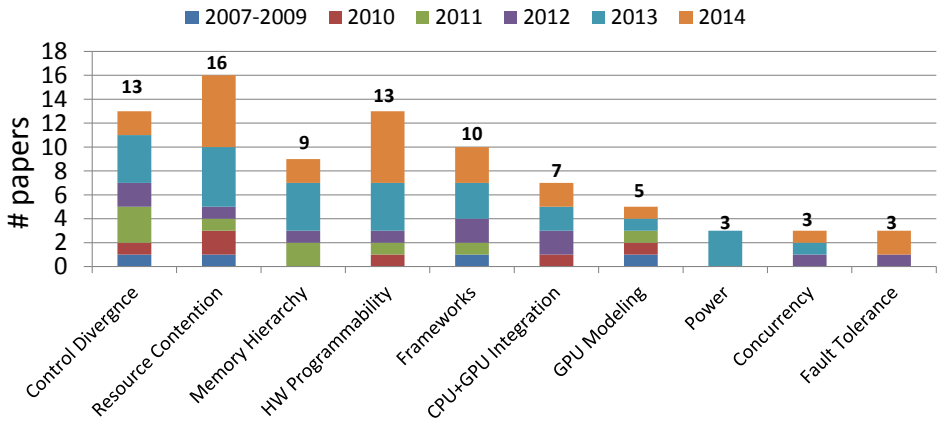

Figure 3.2: Characterization of research papers related to GPGPU shown in figure 3.1

techniques to improve the GPGPU concurrency to increase the thread level parallelism (i.e number of active warps at a time) and efficiently utilize the SM resources. As the number of transistor doubles every 18 months (according to Moore's law), chip die resources also increases over time. To amortize the increasing chip die, there have been some efforts last years, from both academia and industry, to integrate CPU with GPU on chip. Such designs need to be taken into account and carefully studied. For instance, GPUs execute hundreds of threads that can monopolize on-chip shared resources (e.g. memory, on-chip network and last level cache) and this leads CPU applications to be starved. To address this problem, researches have worked on efficiently and fairly managing shared resources between CPU and GPU. We will discuss the state-of-the-art work of each research area in the following sections.

3.2 Improving GPGPU Performance

There are three main bottlenecks that limit GPGPU performance [14, 15] and they are:

1. Control Flow Divergence: Control flow divergence occurs when threads in the same warp execute different control flow paths. Current GPUs employ PDOM mechanism (discussed in chapter 2) that serialize the execution of divergent paths. This serialization of divergent control flow paths reduces thread level parallelism (i.e. the number of active warps at a time). Also, control divergence reduces the number of active threads in warps when they execute, as a result the SIMD execution units are not fully utilized when diverged warp is launched. It has been shown that highly-divergent applications causes SIMD execution units to be underutilized for 50% of the time [42].

2. Memory Bandwidth: Due to the massive multithreading, GPGPU caches and memory hierarchy suffer from severe resource contention which may degrade the performance. Memory divergnce is the main source of GPU resource conention (espacailly cache conention). Memory divergence occurs when threads in the same warp access different regions of memory in the same SIMT instruction. Moreover, as we discussed earlier, GPUs are throughput-oriented architecture, thus the GPUs memory hierarchy should be aware of that. Designing a many-thread aware memory hierarchy is needed to sustain the high memory bandwidth demands of GPGPU applications.

3. Available Parallelism: GPUs achieve high performance by running many concurrent threads on their massively parallel architecture. However, some applications have a low number of active thread blocks due to the small input size or the unavailability of resources in SM (e.g. registers or shared memory), thus they fail to efficiently utilize the execution units.

   In the following sections, we discuss the research work that have been done to alleviate control flow divergence, mitigate resource contention, boost memory bandwidth, and increase parallelism.

3.2.1 Alleviating Control Flow Divergence

Overview

For the purpose of this study, we classify the techniques that address the control flow divergence problem into the following categories:

1. Micro-architectural approaches:

   - Dynamically regrouping divergent warps [16, 42, 88]
   - Large Warps/CTA Compaction [41, 111, 128, 130, 151]
   - Multi-path execution [37, 108, 129, 155]
   - Stack-less architecture [29, 97]
   - Dynamic Micro-kernels [144]

2. Compiler-based and software approaches: [30, 35, 53, 54, 96, 97, 136, 152, 166, 168]

Discussion

Fung et al. [42] proposed Dynamic Warp Formation (DWF) that is not restricted to PDOM reconvergence. Instead, it opportunistically groups threads that arrive at the same PC, even though they belong to different warps. DWF performance is highly dependent on the scheduling to increase the opportunity of forming denser warps, and sometimes leads to starvation eddies.

Fung et al. [41] proposed Thread Block Compaction (TBC) that allows a group of warps to share the same SIMT stack. Hence, at a divergent branch, threads from grouped warps are compacted into new more dense warps. Since TBC employs a thread block wide stack, it suffers more from the reduced thread level parallelism.

Dual-Path stack (DPS) [129] was proposed to support two concurrent paths of execution while maintaining reconvergence at immediate post-dominators. Instead of stacking the taken and not-taken paths one after the other, the two paths are maintained in parallel. DPS maintains separate scoreboard units for each path to avoid false dependencies between independent splits. However, it is necessary to check both units to make sure there are no pending dependencies across divergence and reconvergence points.

Dynamic Warp Subdivision (DWS) [108] was proposed that adds a warp splits table to the conventional stack. Upon a divergent branch, it uses heuristics to decide which branches start subdividing a warp into splits and which do not. If a branch subdivides a warp, DWS ignores IPDOMs nested in that branch. This often degrades DWS performance compared to the PDOM model.

Simultaneous Branch Interweaving (SBI) [16] was proposed to allow a maximum of two warp splits to be interleaved. However, SBI targets improving SIMD utilization by spatially interleaving the diverged warp splits on the SIMD lanes. The reconvergence tracking mechanism proposed with the SBI requires constraints on both the code layout and the warp splits scheduling priorities to adhere to thread-frontier based reconvergence.

Wang et al. [29] proposed Multiple SIMD Multiple Data (MSMD) that requires quite large changes to the baseline architecture to support flexible SIMD datapaths that can be repartitioned among multiple control flow paths.

3.2.2 Efficient Utilization of Memory Bandwidth

Overview

For the purpose of this study, we classify the techniques into the following categories:

1. Alleviating cache thrashing and resource contention:
    - Warp throttling [98, 132, 133, 171]
    - CTA throttling [45, 71, 73, 95, 111, 142]
    - FIFO buffers [69]
    - Cache management scheme [25, 26, 34, 98, 102]
    - Software approach [68, 107, 161]
2. Reducing Memory-Overfetching and Off-Chip Memory Traffic

- Adaptive access granularity [131]
- Data compression [137]
- L1 Cache sharing [148]
- Compiler-based Approach: Cache Modifiers [68] [27], Data Packing Approximation [135]

3. Reducing Memory Divergence and Long Latency Memory Access
   - Memory miss tolerance [45, 147]
   - Two-level thread scheduler to reduce long latency memory operation [45, 111]
   - Compiler-based Approach: [24, 136, 158]
4. Designing Many-Thread Aware Memory Hierarchy
   - Memory Prefetching [7, 87, 91, 99, 102, 138]
   - Interconnection Network [11, 12, 79, 79]
   - Memory Scheduling: [10, 20, 70, 81, 86, 167]
5. Reducing CPU-GPU Memory Transfer [50, 82, 83, 103]

Discussion

Jog et al. [71] proposed CTA-aware-locality scheduling that gives a group of CTAs higher priority to keep their data in the L1 cache such that they get the opportunity to reuse it. Kayiran et al. [73] proposed dynamic CTA scheduling, which attempts to allocate optimal number of CTAs per-core in order to reduce contention in the memory sub-system. Lee et al. [95] explored two alternative thread block scheduling schemes. Lazy CTA scheduling was proposed to leverage GTO scheduler to determine the optimal number of CTAs per core. They also showed how block CTA scheduling (BCS), where consecutive thread blocks are assigned to the same cores, can exploit inter-block locality (i.e. intra-core and inter-core locality).

Rogers et. al. [132] proposed Cache Conscious Wavefront Scheduling (CCWS) to alleviate cache thrashing. CCWS uses a victim tag array, called lost locality detector, to detect warps that have lost locality due to thrashing. These warps are prioritized till they exploit their locality while other warps are descheduled (not allowed to issue any load instructions). Later, Rogers et al. introduced a follow-up work and proposed Divergence-Aware Warp Scheduling (DAWS) [133]. DAWS is a divergence-based cache footprint predictor to estimate the amount of locality in loops required by each warp. DAWS uses these predictions to prioritize a group of warps such that the cache footprint of these warps does not exceed the capacity of the L1 cache.

Li [98] observed that throttling techniques leave memory bandwidth and other chip resources (L2 cache, NOC and EUs) significantly underutilized. Thus, he proposed a cache bypassing scheme on top of CCWS, called Priority-based Cache Allocation (PCAL). PCAL starts from an optimal number of active warps, that alleviates thrashing and conflicts, then extra inactive warps are allowed to bypass cache and utilize the other on-chip resources. Thus, PCAL reduces the cache thrashing and effectively employs the chip resources that would otherwise go unused by a pure thread throttling approach.

A similar approach was proposed by Zheng et al. [171], called Adaptive Cache and Concurrency (CCA). CCA improves DAWS by allowing extra inactive warps and some streaming memory instructions from the active warps to bypass the L1 cache and utilize on-chip resources.

Hawa et al. [71] proposed Memory Request Prioritization Buffer (MPPB). The idea of MRPB is two-fold. First, a FIFO requests buffer is used to reorder memory references so that requests from the same warp are grouped and sent to the cache together and thus reducing the number of warps that access the cache at a time. Second, MRPB allows memory request that encounters associativity stall to bypass L1 cache.

Chen et al. [26] proposed G-Cache to alleviate cache thrashing. To detect thrashing, the tag array of L2 cache is enhanced with extra bits (victim bits) to provide L1 cache by some information about the hot lines that have been evicted before. An adaptive cache replacement policy is used by L1 cache to protect these hot lines. Chen [25] continued his work and proposed Coordinated Bypassing and Warp Throttling (CBWT). CBWT adopts a thrashing-resistant CPU cache management scheme, Protection Distance Prediction (PDP) [36], to GPU cache. PDP employs cache bypassing to enable protection on hot cache lines and thus alleviate cache thrashing. Excessive bypassing may over-saturate the on-chip network. Therefore, cache bypassing policy is coordinated with a dynamic warp throttling mechanism to avoid over-saturating on-chip resources.

3.2.3 Increasing Parallelism and Concurrency

Overview

For the purpose of this study, we classify the techniques into the following categories:

1. Reducing Resource Fragmentation [46, 159, 163]
2. Multitasking (Concurrency Support)
    - Software Spatial Multitasking: [1, 18, 49, 120, 154, 172]
    - Hardware Spatial Multitasking (Resource Management): Cores partitioning [51] Mixed Concurrent Kernel [9, 95] Variation-aware partitioning [2], QoS-aware [4], Fair Memory System [71]
    - Preemptive Multitasking: [146]

Discussion

Gebhart et al. [46] proposed a unified local memory which can dynamically change the partitioning among registers, cache, and scratchpad on a per-application basis. The tuning that this flexibility enables improves both performance and energy consumption, and broadens the scope of applications that can be efficiently executed on GPU.

Xiang et. al. [163] discussed the problems due to thread block level resource management. They classified the resource under utilization problems as temporal and spatial. Temporal under utilization is caused due to di fferences in run times of warps of a thread block, whereas, spatial under utilization is caused because of unavailability of enough resources for a complete thread block at the time of launching a kernel. They proposed a hardware solution to launch a partial thread block when there are not enough resources to launch a full thread block.

Tanasic et. al [146] proposed a set of hardware extensions that allow GPUs to efficiently support multiprogrammed GPU workloads. They design two preemption mechanisms that can be used to implement GPU scheduling policies. They extend the architecture to allow concurrent execution of GPU kernels from different user processes and implement a scheduling policy that dynamically distributes the GPU cores among concurrently running kernels, according to their priorities.

Recent work [9, 95] proposed Mixed Concurrent Kernel/Application Execution, in which two applications execute concurrently on the same core. They showed how such sharing execution can improve the overall system throughput, especially mixture of memory-intensive and compute-intensive workloads. In this mixture, the compute-intensive workload's warps hide the memory latency of memory-intensive workload's warps and thus efficiently utilize the execution units.

3.3 Enhancing GPGPU Programmability

Overview
GPGPU programming is hard and complex. Prior work have explored new techniques to enhance GPGPU programmability. In fact, most of these works were about addressing the same challenges that were found in CPU multi-core programming (i.e. cache coherence, consistency, synchronization, and transactional memory). However, this is not a trivial task as GPUs run thousands of threads concurrently, while CPUs run 4-16 threads. Building a scalable hardware to enhance GPGPU programmability is a key challenge. We classify these works that aim to improve GPGPU programmability into the following categories:

1. Transactional Memory (TM)
    - Hardware TM [43, 43]
    - Software TM [19, 162]
2. Coherence and Consistency
    - Cache Coherence: Hardware [57, 140], Compiler-based [119]
    - Consistency [59, 63, 143]
3. Synchronization
    - Hardware Synchronization Acceleration: [40, 165]
    - Software-based Synchronization: Optimization [38, 160], Approximation [135], Primitives[145], Lock-free Data Structure [17, 110]
4. Memory Management, Virtual Memory and Exception Support:
    - Exception Support [80, 109]
    - Hardware Memory Management and Virtual Memory [82, 83]
    - Software-based Virtual Memory [67]
5. Deterministic and Data Races:

- Hardware Approach: Deterministic [72], Data Race detection [62]
- Software Data Races Detection: [169, 170]

Discussion

Fung et. al. [43] proposed KILO TM, a novel hardware Transactional Memory (TM) design for GPUs that scales to 1000s of concurrent transactions. Without cache coherency hardware to depend on, it uses word-level, value-based conflict detection to avoid broadcast communication and reduce on-chip storage overhead. It employs speculative validation using a novel bloom filter organization to increase transaction commit parallelism. Enabling transactional memory on GPUs simplifies synchronization, and provides a powerful programming model that promotes fine grained communication and strong scaling of parallel workloads.

GPUs lack cache coherence and require disabling of private caches if an application requires memory operations to be visible across all cores. Coherence greatly simplifies supporting well-defined consistency and memory models for high-level languages on GPUs. However, GPU runs 1000s threads concurrently which requires high coherence traffic overheads and impractical amount of storage for tracking thousands of in-flight coherence requests. To enable a scalable and practical GPU cache coherence, Singh et al. [140] proposed a time-based coherence framework for GPUs, called Temporal Coherence (TC), that exploits globally synchronized counters in single-chip systems to develop a streamlined GPU coherence protocol. Synchronized counters enable all coherence transitions, such as invalidation of cache blocks, to happen synchronously, eliminating all coherence traffic and protocol races.

Youngsok et al. [82] proposed GPUdmm, a novel high-performance and memory-oblivious GPU architecture using dynamic memory management. GPUdmm enables dynamic memory management for discrete GPU environments by using GPU memory as a cache of CPU memory with on-demand CPU-GPU data transfers. The benefits of GPUdmm are three-fold. First, GPUdmm provides programmers a view of the CPU memory-sized programming space. Second, It significantly reduces GPU memory size requirements while maintaining target performance. Third, GPUdmm effectively overlaps GPU executions and CPU-GPU data transfers.

Jooybar et. al. [72] proposed a scalable hardware mechanism, GPUDet, to provide determinism in GPU architectures to ease debugging and testing of GPU applications. Deterministic GPUs is essential to enable a broader class of software to use GPUs. GPUDet leverages the inherent determinism of the SIMD hardware in GPUs to provide determinism within a wavefront at no cost. GPUDet also exploits the Z-Bu ffer Unit, an existing GPU hardware unit for graphics rendering, to allow parallel out-of-order memory writes to produce a deterministic output.

3.4 CPU-GPU Heterogeneous Architecture

Overview

An on-chip heterogeneous architecture that integrates GPU cores on top of conventional CPU-only chip multiprocessors (CMP) has become a popular architecture trend, as can be seen in Intels Haswell [64] and AMDs accelerated processing units (APU), like AMD Fusion Kaveri [6], and NVIDIA's Denver project [115]. In this architecture, the concurrent

CPU and GPU applications will share most of the on-chip resources (last level cache, interconnection network and memory controller). However, GPUs run thousands of active threads concurrently, while CPUs run between 4-16 threads. The high degree of thread-level parallelism (TLP) in GPU cores leads to more frequent memory requests. Therefore, a fairness-aware scheme is needed to manage the concurrent applications and avoid GPU applications to monopolize the available resources. Further, when we integrate CPU and GPU on chip, some sort of programmabilty is required to improve data sharing between CPU and GPU cores (i.e. coherence and unified virtual memory address).

For the purpose of this study, we classify the techniques that aim to improve CPU-GPU integration into the following categories:

1. Shared Resources Management
    - Interconnection Network: [92, 93]
    - Last Level Cache Management: [90, 105]
    - Main Memory Scheduling: [8, 66]
    - Throttling-based technique: [74]
2. CPU-GPU Programmability
    - Cache Coherence and Consistency: [58, 85, 122, 124]
    - Unified Virtual Memory: [123, 125]
3. CPU-GPU Assistance:
    - CPU-Assisted GPGPU: [164]
    - GPU-Assisted CPU: [94, 134, 157]

Discussion

Rachata et al. [8] proposed Staged Memory Scheduling (SMS), a decentralized architecture for application-aware memory scheduling in the context of integrated multi-core CPU-GPU systems. SMS is a fundamentally new approach that decouples the memory controllers three primary tasks into three significantly simpler structures that together improve system performance and fairness, in integrated CPU-GPU systems.

Lee et al. [90] proposed a thread-level parallelism (TLP) aware cache management policy for CPU-GPU systems. Due to the presence of deep-multithreading, a cache policy does not directly affect the performance in GPUs. By having a huge number of threads and continuing to switch to the next available threads, GPGPU applications can hide some of the off-chip access latency. However, some GPGPU applications are cache-sensitive and are not able to tolerate long memory latency. Hence, to estimate the effect of cache behavior on GPU performance, they propose a core-sampling approach, which applies a different policy (e.g. a cache replacement policy) to each core and periodically collects samples to see how the policies work. A large difference in performance of these cores indicates that GPU performance is affected by the cache policy and vice versa. Further, GPGPU applications typically access caches much more frequently than CPU applications. To ensure fairness and alleviate the interference, they introduced cache block lifetime normalization approach, which ensures that statistics collected for each application are normalized by the access rate of each application.

Kayiran et al. [74] proposed GPU concurrency management that dynamically throttling/boosting TLP (i.e. number of active warps) of GPU cores in order to minimize shared resources interference between CPU and GPU.

Power et al. [58] presented a framework for supporting directory-based hardware coherence between CPUs and GPUs in a heterogeneous CPU-GPU system. They assume a heterogeneous system where CPU and GPU clusters have two separate, non-inclusive, shared L2 caches. Their coherence scheme replaces a standard directory with a region directory and adds region buffers to L2 caches of both CPU and GPU to track the regions over which the CPU or GPU currently hold permission. These structures allow the system to move the coherence-related traffic from the coherence network to the high-bandwidth direct-access bus while still maintaining coherence.

Chapter 4: Experimental Setup and Workload Characterization

In this chapter we describe our experimental setup. First, Section 4.1 shows the configuration parameters of our simulator and the workloads we used in this study. Second, we analyze and characterize the cache behaviors of our GPGPU workloads. We show the workload characterization methodology and results in section 4.2 and 4.3 respectively.

4.1 Experimental Methodology

We simulate the baseline architecture using GPGPU-Sim v3.2.1 [13], a publicly-available cycle-accurate GPGPU simulator. The GPU simulator is configured to be similar to NVIDIA Fermi GTX480 [156]. We use the configuration file provided with GPGPU-Sim without any modifications. The configuration parameters are described in Table 4.1. The simulator was modified to implement the proposed techniques that we evaluate in this work.

We considered a wide range of GPGPU CUDA workloads, including applications from Rodinia [21], Poly-Bench [48] and NVIDIA SDK [113]. NN, IIX, SPMV_S, and KM are adopted from GPGPU-sim workloads [13], MapReduce [56], SHOC [32], and CCWS applications suite [132] respectively. In total, we study 22 applications described in Table 4.2. The applications run until completion, with the exception of SYRK, GESUMMV, and SCLUSTER, due to the long simulation time of these applications, we execute SYRK and GESUMMV only up to 100 million instructions, while SCLUSTER up to 300 million instructions.

4.2 Workload Characterization Methodology

In order to understand the cache sensitivity of GPGPU applications, we run our workloads in three different cache scenarios: (1) totally bypassing all memory accesses (i.e. no caches), (2) Bounded caches using the baseline configuration (16KB L1, 786KB L2) and (3) A fully-associative unbounded caches (only cold misses occur in this scenario and it represents the upper bound of performance improvement from using caches). Note that, in unbounded caches, the other cache resources (e.g. Miss Status Handling Registers, MSHRs) are still limited as the baseline configuration and not increased. The results are shown in Figure 4.2. Further, we analyze and measure the amount of locality that exist in GPGPU workloads in the bounded and unbounded scenarios. We classify locality that occurs in GPPGU application to the following five categories:

(1) *Intra-warp* data locality occurs when a cache line is referenced and re-referenced by the same warp. we can further divides the intra-warp locality into two subcategories (intra-thread and inter-thread locality [132]). Intra-thread occurs when a cache line is

Table 4.1: Simulated baseline GPGPU configuration

SM configuration	15 SMs, 700 MHZ, 1536 threads, 32 threads/warp, 48 warp/SM, SIMD width = 32, 5-Stage Pipeline, 32684 registers
L1 Cache	16KB/4-way associativity/128B cache line/global-write-evict-local-write-back/no-write-allocate/allocate-on-miss/32 MSHR entries
L2 Cache	6 partitions x 128KB/16-way/128B cache line/write-back/write-allocate/ 32 MSHR entries
Shared Memory	48 KB
Constant Cache	8KB
Texture Cache	12KB/24-way/128B line
# Warp scheduler	2 per SM (24 warps per scheduler)
Warp scheduling	Greedy-then-oldest (GTO) [132]
Branch Divergence	PDOM [42]
Interconnect	1 crossbar/direction, 32B channel width, 1400 MHZ
Memory Model	6 GDDR5 Memory Controllers (MCs), First-Ready FCFS (FR-FCFS) scheduling, 924 MHz, BW=179.2 GB/s
GDDR5 Timing	tCL=12, tRP=12, tRC=40, tRAS=28, tRCD=12, tRRD=6

Table 4.2: GPGPU Workloads

Name	Abbrev.	Type
Black Scholes [113]	BLK	Streaming L1/L2
Scalar Product [113]	Sprod	Streaming L1/L2
Vector Addition [113]	VAdd	Streaming L1/L2
Fast Walsh Transform[113]	FWT	Large working set L2
Needleman-Wunsch [21]	NW	Large working set L2
Hot Spot [21]	HS	Streaming L1
Separable Convolution [113]	CONV	Streaming L1
Structured grid [21]	SRAD	Conflict
3D Stencil [21]	3DS	Conflict
2D Convolution [48]	2DCONV	Conflict
2 Matrix Multiplication [48]	MM	Conflict
Stream Cluster [21]	SCLUSTER	Conflict
Breadth First Search [21]	BFS	Thrashing
Sparse Matrix Vector Multplication[32]	SpMV	Thrashing
Inverted Index [56]	IIX	Thrashing
Kmeans Clustering [132]	KM	Thrashing
Symmetric Rank-k [48]	SYRK	Thrashing+Conflict
Vector Matrix Multiply [48]	GESUMMV	Thrashing+Conflict
MCARLO Pi Estimator [113]	PEst	Friendly
B+tree [21]	B+tree	Friendly
Back Propagation [21]	BP	Friendly
Neural Network [13]	NN	Friendly

referenced and re-referenced by the same thread, while inter-thread locality occurs when a cache line is referenced and re-referenced by two different threads within the same warp.

(2) *Intra-block* data locality occurs when a cache line is referenced and re-referenced by two different warps within the same thread block.

(3) *Intra-core* data locality occurs when a cache line is referenced and re-referenced by two different warps with different thread blocks, and the two thread blocks are assigned to the same core.

(4) *Inter-core* data locality occurs when a cache line is referenced and re-referenced by two different warps with different thread blocks, and the two thread blocks are assigned to different cores. Obviously, this locality is only exploited through L2 cache.

(5) *Inter-kernel* data locality occurs when a cache line is referenced and re-referenced by two different warps with different kernels. For instance, a data was written by a kernel and a consecutive kernel accesses this data.

Figure 4.3 show the amount and type of locality in L1 and L2. The left bar represents the locality found in unbounded caches while right bar for bounded caches. The L2 cache was evaluated while L1 cache is bounded.

GPGPU workloads exhibit a high level of contention at the few available L1 cache resources (e.g. MHSR entries, cache lines and Miss_Queue entries). Figure 4.1 depicts the process for handling a cache miss. When a cache miss occurs, the miss status handling registers (MSHRs) are checked to see whether the same request has already been issued for another warp and is still pending. If the request was found, a MSHR_MIX entry is allocated to ensure the returned request services both warps. If the request wasn't found in MSHRs, an empty MSHR entry is allocated, a cache line is reserved and a read memory request is placed in the Miss_Queue. However, a cache controller may fail to service a miss request due to lack of any requested resource as shown in Figure 4.1 (MSHR_MIX_ENTRY_FAIL, MSHR_ENTRY_FAIL, LINE_ALLOC_FAIL or MISS_QUEUE_FULL). In this case, the memory request causes the pipeline to stall and it will retry the next cycles until all the requested resources are available. Figure 4.4 depicts the L1 reservation fails per kilo cycles for our workloads and Figure 4.5 shows the misses per kilo cycles (MPKI) [1].

4.3 Workload Characterization Results

From experimental results (shown in Figure 4.2, 4.3, 4.4 and 4.5), we classify our applications into three main categories:

(1) Streaming Applications:

We observe that there are applications that don't benefit from caches at all (BLK, Sprod, and VAdd). In these applications, caches hurt the performance instead. As shown in Figure 4.2, using bounded or unbounded caches leads to loss in performance compared to bypassing. These applications show a streaming behavior in both L1 and L2. They exhibit a high miss rate (up to 99%) in bounded and unbounded caches. For these applications, it is better to bypass their memory accesses since they don't benefit from caches and cause useless unintended contention at L1 (see Figure 4.4).

[1] MPKI is calculated by (number of misses/number of committed instructions)*1000

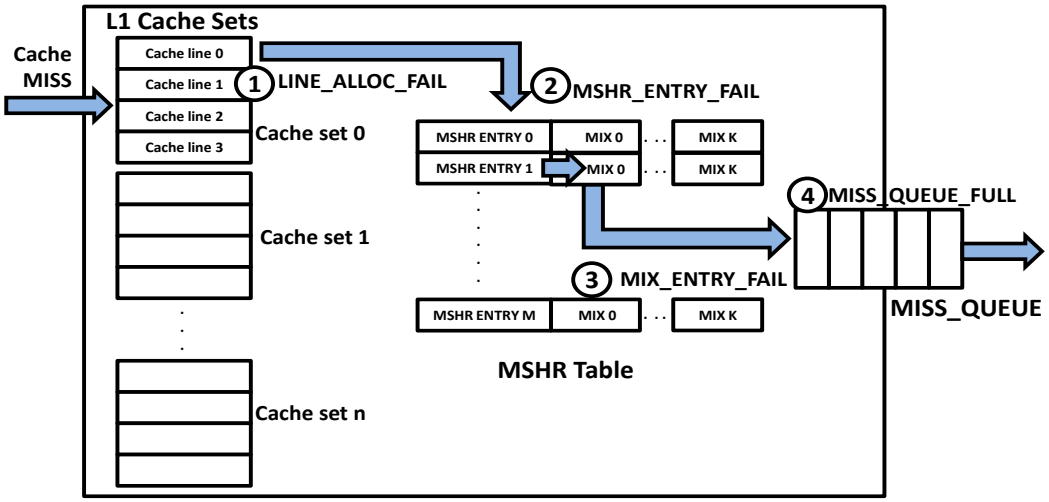

Figure 4.1: L1 Cache Miss Handling. (1) Finding available cache line at the mapped cache set. (2) Searching for MSHR Entry of the same memory address, otherwise allocate a new MSHR ENTRY (3) If MSHR is found, allocate a MIX ENTRY (4) Place a memory request in MISS QUEUE

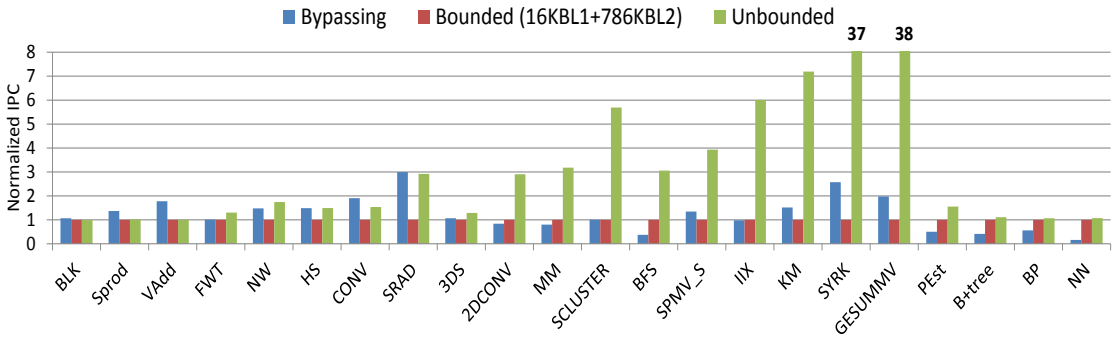

Figure 4.2: Cache Sensitivity

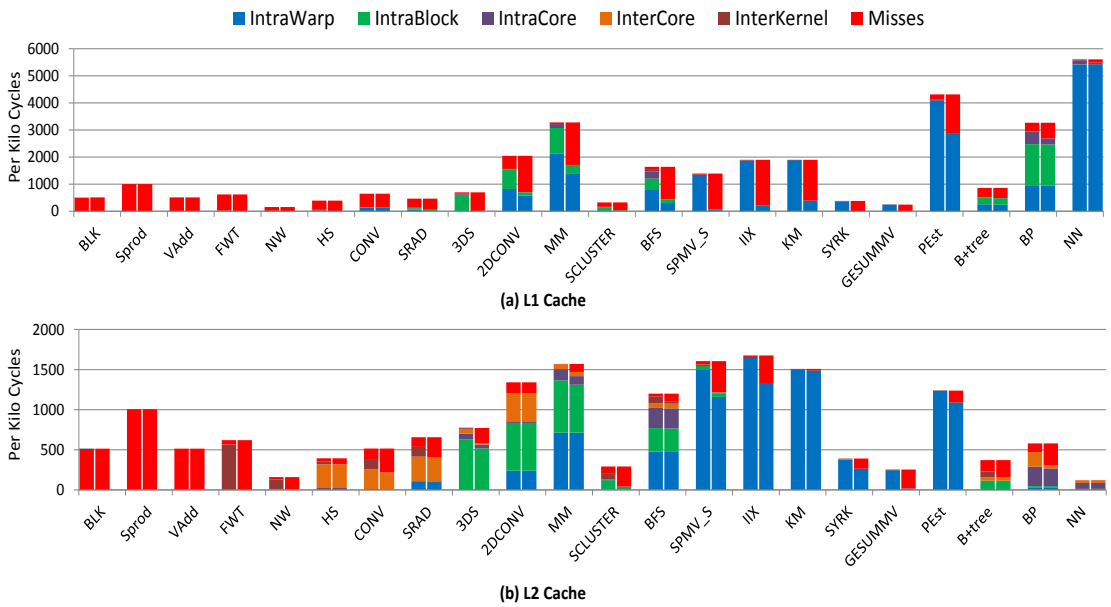

Figure 4.3: L1/L2 Data Locality Analysis. The left bar represents the locality found in unbounded caches while right bar for bounded caches

For (FWT and NW), the unbounded cache is better than bounded and bypassing. These applications also show streaming behavior in bounded and unbounded L1 cache. In contrast, they are full of inter-kernel locality at unbounded L2, however bounded L2 is not large enough to cache the data transferred between kernels. In this work, we bypass these applications, and we leave improving L2 cache to exploit inter-kernel locality for future work.

For (HS and CONV), they show a streaming behavior in L1, while they exhibit a high hit rate at L2. They are full of inter-core locality and thus they are L2 cache sensitive. For these applications, it is better to bypass L1 cache only. If we inspect the code of these workloads, we will find that they rely on the on-chip scratchpad memory to exploit locality. Thus, when they are cached, they show streaming behavior in L1. It is worthwhile to note that, an application that uses scratchpad, doesn't necessarily show streaming behavior in L1. For instance, BP relies on scratchpad, however it is full of locality.

(2) Cache Contention Applications: For these workloads, the unbounded cache is better than bounded and bypassing by an order of magnitude (as shown in Figure 4.2 for SCLUSTER, IIX, SYRK and others). These workloads are full of data locality at the unbounded L1 (as shown in Figure 4.3), however the limited size of bounded L1 cache and the large number of threads GPGPU executes concurrently makes it susceptible to conflict and capacity misses [61, 69].

Conflict misses mainly occur when a group of warps (Inter-warp conflict contention) or a group of threads within the same warp (Intra-warp conflict contention [69]) access the same cache set within a short period of time. The warps/threads compete on the few

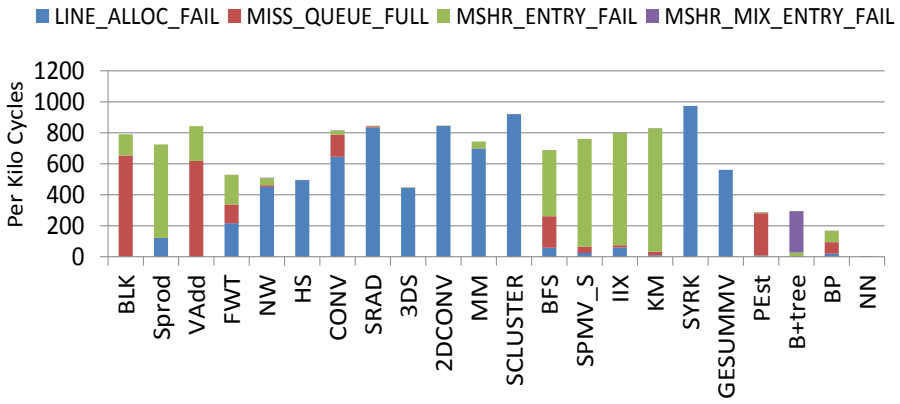

Figure 4.4: L1 Cache Resource Contention

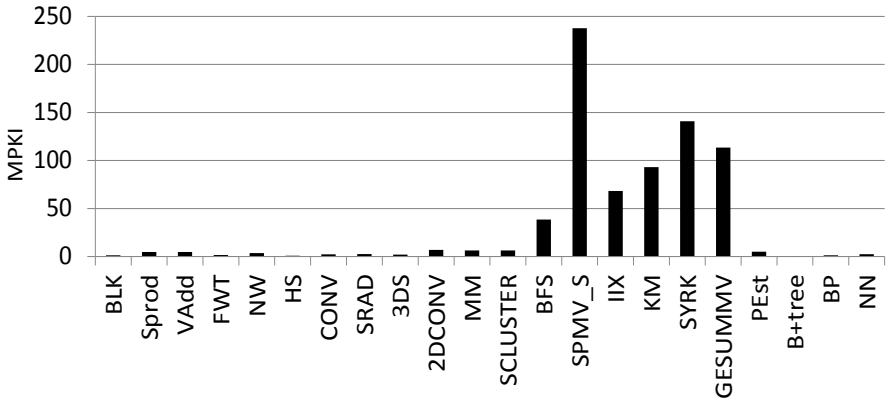

Figure 4.5: L1 Misses Per Kilo Instructions

available cache lines in cache set (typically between 4-6 lines [106]). Consequently, it causes a high level of LINE_ALLOC_FAIL stalls (associativity stalls). Increasing the associativity of L1 cache is able to alleviate this type of contention [69]. Capacity misses mainly occur when the cache footprint per-warp is large (Inter-warp capacity contention). In this case, with no conflict contention and L1 cache cannot fit all the running warps working set, the warps will compete on the cache lines and the cache is susceptible to thrashing. Cache thrashing causes a high level of MSHR_ENTRY_FAIL stalls and MPKI. Increasing the L1 cache capacity is able to alleviate this type of contention [69]. It has been shown that the code style has an effective role to alleviate/increase cache contention [69]. Writing a highly-divergent non-optimized code (mainly programs that contain loops) may cause a severe cache contention [133].

The applications (SRAD, 3DS, 2DCONV, MM, and SCLUSTER) suffer from inter-warp conflict contention. As shown in Figure 4.3, they are full of intra-warp and intra-block locality. However, the bounded L1 is not able to exploit these localities, especially intra-block locality. Figure 4.4 illustrates that these applications exhibit a high level of LINE_ALLOC_FAIL stalls, which means that inter-warp conflict contention occurs in these workloads.

The applications (BFS, SPMV, IIX, and KM) suffer from inter-warp capacity contention. They contain a large intra-warp locality. Most of these locality are intra-thread (not shown in figure). The running warps evict the cache lines of each other's and cause severe thrashing at L1 (see Figure 4.3) and high levels of MSHR_ENTRY_FAIL stalls (see Figure 4(a)). Figure 4.5 shows that these thrashing applications exhibit a high level of MPKI over other applications. Hence, MPKI can be used as a good measure to detect thrashing.

The applications (SYRK and GESUMMV) suffer from intra-warp conflict contention and inter-warp capacity contention. The LINE_ALLOC_FAIL stalls and MPKI for these applications are high. In section 5.3, we discuss the behavior of these workloads in details. Note that, in thrashing applications, the L2 cache backs up L1 evicted cache lines for future reuse, except for GESUMMV. In GESUMMV, the cache footprint per-warp is large to the extent that thrashing also occurs at L2 (see Figure 4.3).

(3) Cache-friendly Applications: For (Pest, B+tree, BP, and NN), the bounded cache is much better than bypassing and it nearly achieves the same performance of unbounded cache. They exhibit a high hit rate at both L1/L2 and L1 reservation fails is reasonable.

Next Chapter, we describe our proposed methods to alleviate thrashing, aviod conflict misses, and bypass streaming applications.

Chapter 5: Efficient Utilization of GPGPU Cache Hierarchy

In this chapter we describe our proposed methods to bypass streaming behavior, alleviate thrashing and avoid conflicts. This Chapter is organized as follows, Section 5.1 discusses our proposed method to dynamically bypassing memory access that show streaming behavior. Section 5.2 presents, Dynamic Warp Throttling via Core Sampling (DWT-CS), a novel technique to address the thrashing problem of GPGPUs cache. Section 5.3 shows our third technique to alleviate cache conflicts contention. Section 5.4 shows how to combine all the proposed methods together in one algorithm. The remaining of this chapter is devoted to discuss related work.

5.1 Dynamically Bypassing Streaming Applications (DBSA)

To address the streaming behavior problem, we dynamically detect and bypass streaming applications (DBSA). The proposed method monitors the L1 cache miss rate over sampling periods. At the end of each sampling period, it checks whether the miss rate is larger than a specific threshold. If so, the cache is disabled and all the memory accesses bypass the L1 cache. We implement the same method for L2 caches. Some workloads don't show a constant behavior over time. For instance, as shown in 5.1, CONV shows a streaming behavior during the first half of its execution time, then it shows a high hit rate for the second half. To remedy this, our method leaves the cache controller enabled during bypassing. The cache controller updates tags only and calculates the new miss rate. If the miss rate is less than the predefined threshold, the cache is enabled. While our method applies a coarse-grained cache bypassing scheme (i.e. enable or disable the whole cache), we leave building a fine-grained cache bypassing scheme for future work.

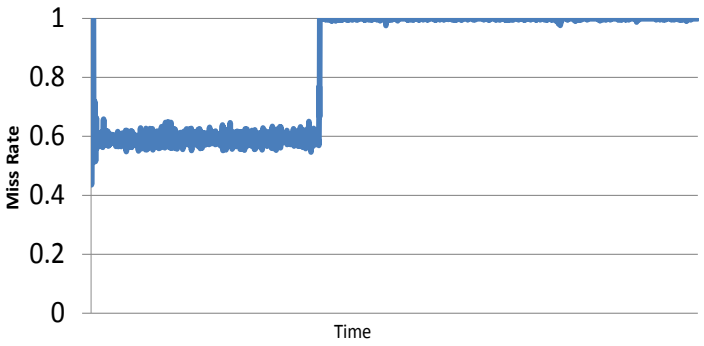

Figure 5.1: CONV miss rate over time

Table 5.1: The number of active warps (per warp scheduler) achieved by SWT vs DWT-CS

Benchmark	SWT	DWT-CS
SPMV	1	1
Kmeans	1	1
BFS	5	7
IIX	2	2
SYRK	2	2
GESUMMV	1	1

5.2 Dynamic Warp Throttling via Cores Sampling (DWT-CS)

Static Warp Throttling (SWT): Prior work [132, 133] proposed warp throttling as an effective method to alleviate the cache thrashing problem. The number of active running warps per core is throttled to a lower number such that their cache footprint can be fitted in cache. Static Warp Throttling (SWT) (a.k.a. Best Static Warp Limiting [132]) statically runs an exhaustive search to find the best number of active warps that achieves the highest performance. All possible warp numbers per warp scheduler (24 to 1 in our case) were tested and the best performing one is selected. Figures 5.2, 5.3, 5.4 and 5.5 show the warp throttling effect on our thrashing workloads (BFS, SPMV, Kmeans and IIX respectively). As shown in figures, when we decrease the number of active warps, the MPKI decreases and the thrashing is alleviated, and thus the performance (i.e. IPC) is increased. It is important to note that each application has different number of active warps that achieves the highest performance. Table 5.1 lists the best number of active warps for each thrashing application. It is also worth noting that the best number of warps doesn't occur at the lowest degree of MPKI. For example, as shown in Figure 5.2, BFS achieves the best performance at five active warps, while the lowest MPKI occurs at one active warp. Also, as shown in Figure 5.5, IIX chieves the lowest MPKI at one active warp while the best performance occurs at two active warps. This is because the best number of warps is a trade-off number between MPKI and Thread-level Parallelism (TLP). If you aggressively throttle the number of active warps to mitigate thrashing, this may affect the TLP as well, and the available warps will not be able to hide memory latency.

Dynamic Warp Throttling (DWT): In contrast, Dynamic Warp Throttling (DWT) aims to find the best number of active warps dynamically by hardware. Cache Conscious Wavefront Scheduling (CCWS) [132] and Divergence-Aware Warp Scheduling (DAWS) [133] are two proposed schemes to implement DWT. CCWS uses a victim tag array, called lost locality detector, to detect warps that have lost locality due to thrashing. These warps are prioritized till they exploit their locality while other warps are descheduled (not allowed to issue any load instructions). DAWS introduced a divergence-based cache footprint predictor to estimate the amount of locality in loops required by each warp. DAWS uses these predictions to prioritize a group of warps such that the cache footprint of these warps don't exceed the capacity of the L1 cache.

Dynamic Warp Throttling via Cores Sampling (DWT-CS): CCWS and DAWS

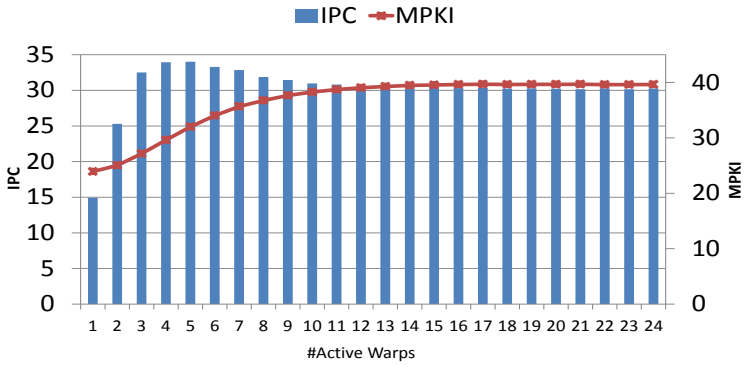

Figure 5.2: BFS warp throttling effect

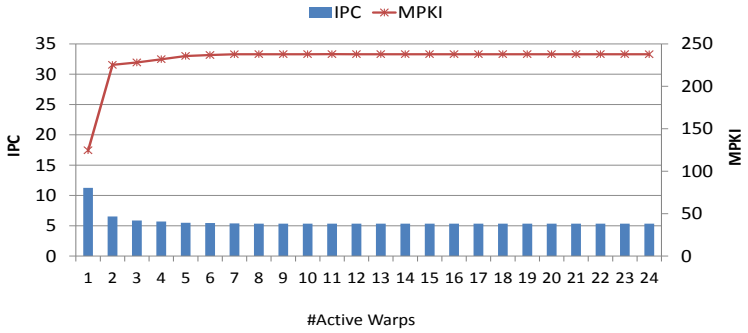

Figure 5.3: SPMV warp throttling effect

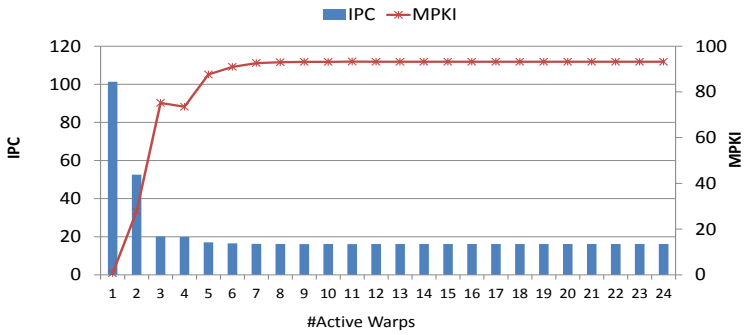

Figure 5.4: Kmeans warp throttling effect

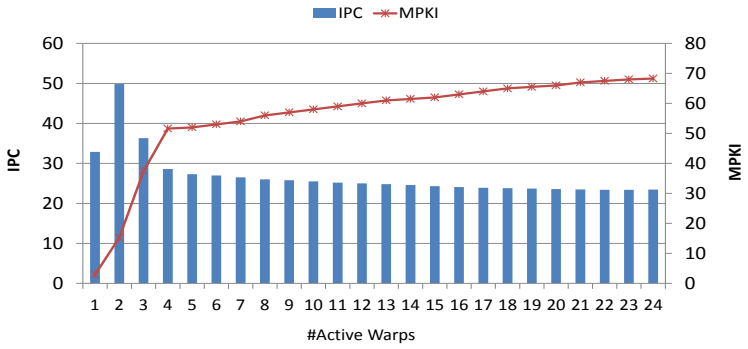

Figure 5.5: IIX warp throttling effect

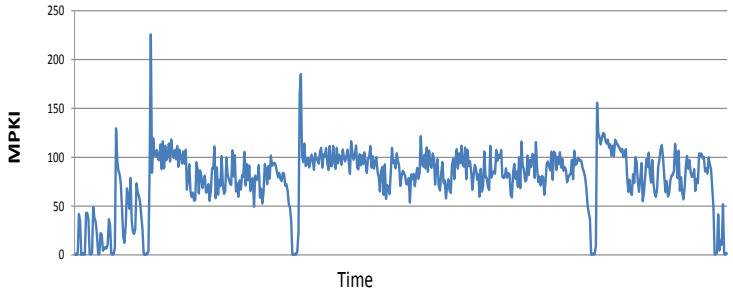

Figure 5.6: BFS MPKI over time

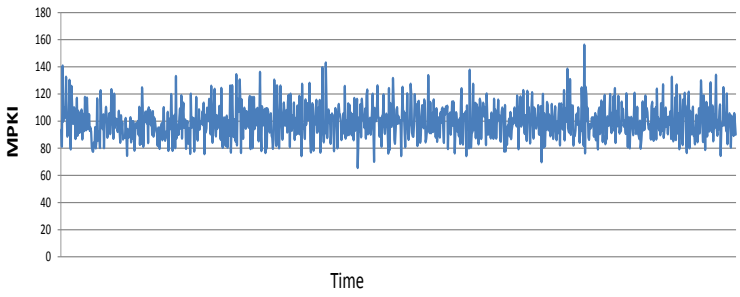

Figure 5.7: Kmeans MPKI over time

have a fine-grained control on warp throttling (i.e. the number of active warps is variable over time depending on the thrashing level). This gives them an advantage over SWT. However, it has been shown that the coarse-grained SWT approach has a comparable speedup to CCWS and DAWS. SWT is able to outperform CCWS and almost achieves the same performance of DAWS on average [132]. This is due to the fact that many GPGPU applications show a consistent thrashing level over time. In addition, SWT tries to find the best trade-off number of warps that works well at different thrashing levels and improves the overall performance. However, SWT is not a practical solution. The programmer needs to do an exhaustive search for each application. Moreover, SWT is input sensitive, which means that the best number of warps changes when running the same application with different input sets [132]. In this work, we propose Dynamic Warp Throttling via Cores Sampling (DWT-CS). DWT-CS uses a similar approach as SWT (i.e. exhaustive searching), however it lets the hardware handle the searching process. Thus, DWT-CS overcomes SWT shortcomings. The idea of core sampling was proposed by Lee et al. [90] by applying different cache management policies to different cores and collecting samples to see how these policies behave. In this work, we employ a similar mechanism to find the best number of active warps that alleviates thrashing and efficiently utilizes L1 cache.

DWT-CS monitors the MPKI at L1 over sampling periods (from Figure 5.15, L1 MPKI can be used as a good measure to detect thrashing). At the end of each sampling period, it checks whether the MPKI has exceeded a specific threshold for N consecutive periods. If so, all GPU cores are sampled with different number of active warps, equals to the core ID (For example, core#1 throttles the active warps to only one warp, core#2: two active warps and so on). After M sampling periods, all cores send the number of instructions committed during the sampling periods to the coordinator core (for example, the middle core, core#8 in our case). The coordinator core finds the core ID (i.e. number of warps) that has executed the maximum number of instructions. The winner core ID is propagated to all the cores. Next, the cores throttle the number of active warps to the new propagated value and it remains till the end of kernel execution. In case, the winner core is the last core (core# 15 in our case), another sampling period is relaunched to test the other numbers of warps (16-24). However, this case did not occur in our benchmarks and it rarely exists. When the same kernel is relaunched and MPKI exceeds the threshold, it doesn't sample the cores again. Instead, it uses the same number of warps obtained before.

Table 5.1 shows the best number of active warps achieved by SWT and DWT-CS. The DWT-CS is able to achieve the same number of warps as SWT for all benchmarks except for BFS which consists of small kernels and suffers from phased execution (i.e. non-steady thrashing level). Figure 5.6 and 5.7 depict MPKI over time (per sampling periods) for BFS and Kmeans respectively. In BFS, the thrashing level (MPKI) changes over time in an unsteady fashion. It shows a non-stable thrashing level. The MPKI standard deviation of BFS over time is 31. On contrast, Kmeans shows near-steady thrashing level over time. The MPKI standard deviation of Kmeans is 17 (2x lower than BFS). Fixing the number of active warps, as DWT-CS does, for steady thrashing level (like Kmeans and IIX) works well and can achieve better performance than other proposed approaches (like CCWS and DAWS) since DWT-CS doesn't waste overhead to detect thrashing over time and adjust number of active warps based on thrashing level. The best number of active

warps for these applications is constant. On contrary, applying DWT-CS for non-steady thrashing applications (like BFS and SPMV) is not appropriate. It may achieve a good performance for these applications on average but it is far away from the optimal (see next chapter). The number of warps needs to be changed over time depending on the thrashing level. In this work, we employ a constant number of active warps for non-steady thrashing application and we leave investigating more sophisticated instability-resistant warp throttling mechanism for future work.

Implementation Oveahead: Our proposed DWT-CS is a cost-effective method. It is able to slightly outperform CCWS on average over thrashing applications (see Chapter 6), while requiring negligible hardware overhead. CCWS needs extra hardware (victim tag arrays) to detect thrashing, whereas DWT-CS needs only a divisor[1] and two counters to calculate the committed instructions [2] and cache misses in order to measure MPKI over sampling periods. A few registers are also needed to save the best number of warps per-kernel for future reuse. Moreover, the coordinator core can use one of the built-in SIMD units that support MAX instruction [116] to find the winner core ID. In addition, a small control circuit, that is composed of a 5-bit register (to hold the number of maximum warps for throttling) and a comparator (to check the current number of ctive warps and the value found in 5-bit resgier), is needed to allow throttling and control the number of active warps per warp scheduler. DWT-CS takes around 60K cycles to detect thrashing and find the optimal number of warps. In real applications, this overhead time can be neglected.

5.3 Pseudo Random Interleaved Caches (PRIC)

The problem of CPU cache associativity has been widely studied in literature. Many works proposed different techniques to improve the cache indexer function that is responsible for interleaving memory accesses over cache sets. Prime modulo interleaving [76, 89], 1-skew storage [55], logical data skewing [139], Xor-based functions [141] and Pseudo Random Interleaving [127] have been proposed, instead of the conventional sequential interleaving, in an attempt to improve cache associativity and avoid conflicts. It has been shown that the Pseudo Random Interleaving Cache (PRIC) is a cost-effective high-performance approach [47, 127, 150]. In this work, we employ PRIC for GPU caches and see how effective they are to alleviate associativity stalls and eliminate conflict misses.

Sequential Interleaving Cache In a sequential interleaving cache consisting of $M = 2^m$ cache sets and a cache line size $B = 2^b$ bytes, a N-bit memory location whose address is A[N-1:0], has cache index of A[m+b-1:b] and a Tag address A[N-1:m+b]. Figure 5.8(a) depicts a simple example of how memory locations are interleaved over cache sets in case of sequential interleaving. In an application that generates a stream of M memory references in a short period of time, with an access stride S, has a n-way

[1] Since the divisor is expensive, we can omit it. Alternatively, we can measure the number of L1 misses over sampling periods of instructions (i.e. every specific number of committed instructions). Thus, the denominator (i.e. committed instructions) is constant over the sampling period and we can compare the number of misses properly.

[2] These counters already exist in GPU to monitor GPU performance

S0	S1	S2	S3	S4	S5	S6	S7
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

(a) Sequential Interleaving

S0	S1	S2	S3	S4	S5	S6	S7
0	4	2	6	1	5	3	7
13	9	15	11	12	8	14	10
23	19	21	17	22	18	20	16
26	30	24	28	27	31	25	29
35	39	33	37	34	38	32	36
46	42	44	40	47	43	45	41
52	48	54	50	53	49	55	51
57	61	59	63	56	60	58	62

(b) Pseudo Random Interleaving

Figure 5.8: Memory locations interleaving over cache sets (Assume 6-bit memory address, 1-byte cache line and 8 cache sets, S1 means cache set# 1)

conflict degree where n=M/gcd(M,S), and gcd stands for the greatest common divisor. From this equation, we can observe that even strides will cause a high level of conflict degree. For example, assume the sequential memory interleaving shown in Figure 5.8(a), a reference stream with an access stride of 2 (i.e. 0,2,4,6,8,10,12,14) has a 4-way conflict degree (i.e. all the memory references will be mapped to only 4 cache sets out of 8). Each pair of the addresses (0,8), (2,10), (4,12), (6,14) will map to the same cache set and may cause associativity stalls (in case the cache set contains cache lines less than mapped memory references). The worst case scenario occurs when the reference sequence has a stride which is a multiple of M, thus causing a 1-way conflict where all references map to the same cache set. On the other hand, all odd strides have no common divisor with M greater than one (recall that M is a power-of-2 number) and hence they don't cause any conflicts and the memory references will be distributed evenly over the cache sets. However, it is important to note that even strides, especially strides that are of multiple M, occur frequently in GPGPU applications. For instance, Figure 5.9 shows a GPGPU frequent scenario to access a 2-D matrix. In SYRK, the output element (i,j) is calculated by multiplying row(i) by row(j) of matrix A. The A matrix rows are aligned to the cache line size (i.e. row size=K*line size) and are stored in a row-major order form in the main memory. On each loop iteration, each 32 threads within a warp read 32 elements from different consecutive rows of matrix A. When K is a multiple of the number of cache sets (32 in our baseline), all the memory reference loads will map to the same cache set causing a high level of associativity stalls and conflict misses. Figure 5.10 shows pictorially the behaviour of these threads when they try to access the L1 cache. As shown in figure, the memory-coalescing unit fails to consolidate the 32 threads' loads to a single or few memory loads, thus a load instruction from a single warp causes memory divergence, and up to 32 memory requests (corresponding to 32 threads) are generated at a time. It is important to note that a matrix whose row size equals 32*n*line size, where n>=1, frequently exists in GPGPU applications.

Pseudo Random Interleaving Cache In PRIC, as shown in Figure 5.8 (b), each M

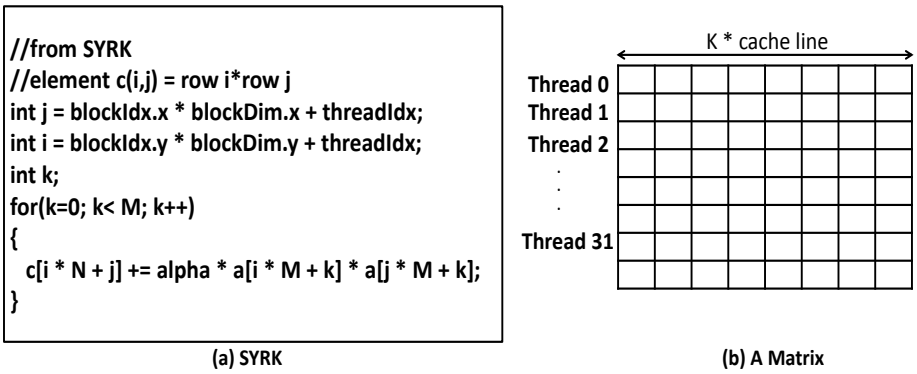

Figure 5.9: An example of 1-way conflict degree in SYRK workload. When K is multiple of the number of cache sets, all 32 threads will map to the same cache set

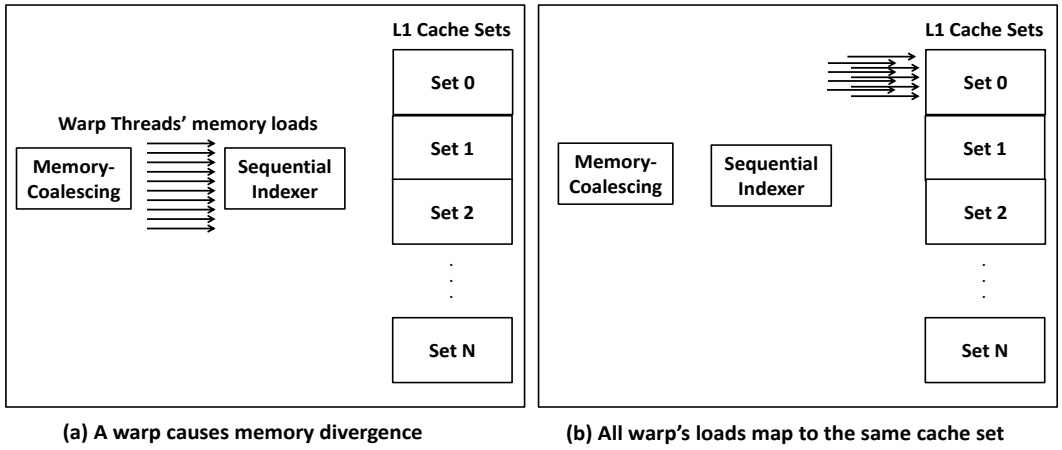

Figure 5.10: Memory divergence causes severe conflict contention in sequential interleaving cache

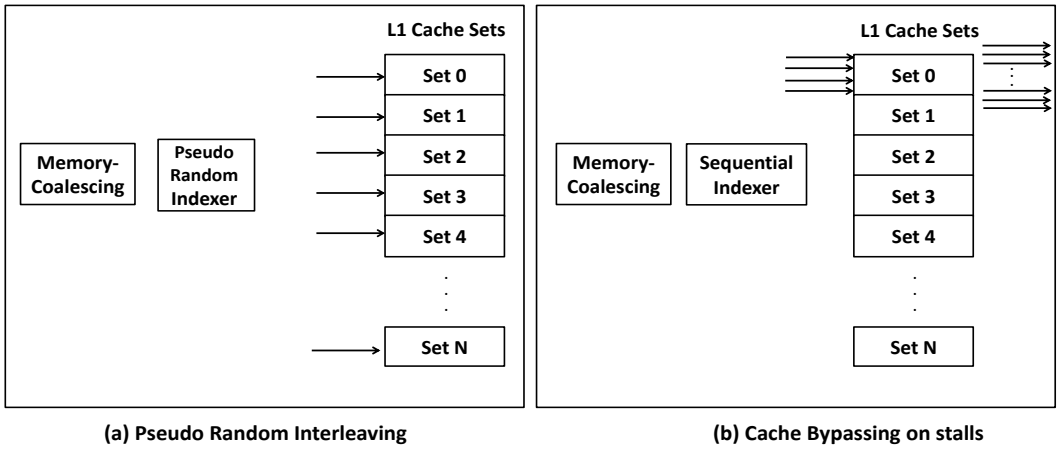

Figure 5.11: Pseudo Random Interleaving vs Cache Bypassing

$$I[4:0] = A[26:7] * \begin{bmatrix} 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 5.12: The H-Matrix corresponding to Poly(37). Cache index I[5:0] is generated by multiplying the address A[26:7] by the H-matrix.

consecutive memory locations have different permutation over the cache sets in a way that make them near-randomly interleaved. This near-random interleaving makes PRIC resistant to all strides, especially strides that are multiple of M. PRIC is based on Polynomial Modulus Mapping in which the memory location, A, is expressed as a polynomial function whose coefficients are in the Galios GF(2). For example, memory location 21 is expressed as $(x^4 + x^2 + 1)$. Let P(x) be a polynomial of order m, and A(x) be the polynomial of order N that is associated with memory location A. Then A(x) can be uniquely represented as A(x) = V(x)*P(x) + R(x) where V(x) and R(X) are polynomials over GF(2) and R(x) is of order less than m. V(x) and R(x) can be seen as the polynomial representation of the corresponding tag and cache index. Hence, the cache index of address R(x) = A(x) mod P(x). It was found that for the best performance and permutation, P(x) should be an Irreducible Polynomial Function (I-Poly). P(x) is said to be irreducible if no two non constant polynomials g(x) and h(x) with rational coefficients such that P(x)=g(x)*h(x) exists [104]. Rau [127] shows how the computation of cache index R(x)=A(x) mod P(x) can be carried out by the vector-matrix product of the address and a matrix of single-bit coefficients, named H-matrix (i.e. I[m-1:0] = A[N-1:b]*H-matrix). In GF(2), multiplication and addition are equivalent to AND and XOR boolean function, and if the matrix is constant, the AND gates can be omitted and the mapping then requires just XOR gates with fan-in from 2 to n [47].

Implementation Overhead In our case, the baseline has $32 = 2^5$ cache sets, thus

$$I_0 = A_{25} \oplus A_{24} \oplus A_{23} \oplus A_{22} \oplus A_{21} \oplus A_{18} \oplus A_{17} \oplus A_{15} \oplus A_{12} \oplus A_7$$

$$I_1 = A_{26} \oplus A_{25} \oplus A_{24} \oplus A_{23} \oplus A_{22} \oplus A_{19} \oplus A_{18} \oplus A_{16} \oplus A_{13} \oplus A_8$$

$$I_2 = A_{26} \oplus A_{22} \oplus A_{21} \oplus A_{20} \oplus A_{19} \oplus A_{18} \oplus A_{15} \oplus A_{14} \oplus A_{12} \oplus A_9$$

$$I_3 = A_{23} \oplus A_{22} \oplus A_{21} \oplus A_{20} \oplus A_{19} \oplus A_{16} \oplus A_{15} \oplus A_{13} \oplus A_{10}$$

$$I_4 = A_{24} \oplus A_{23} \oplus A_{22} \oplus A_{21} \oplus A_{20} \oplus A_{17} \oplus A_{16} \oplus A_{14} \oplus A_{11}$$

Figure 5.13: The Xor-ing equations corresponding to Poly(37)

m=5. There are six irreducible polynomial functions of degree 5 over GF(2) [104] and they are Poly (37, 41, 47, 55, 59, 61). In this study, we empirically use Poly(37). The corresponding H-matrix and Xor-ing boolean equations of Poly(37) are listed in Figure 5.12 and 5.13 respectively [3]. As shown in figure 5.13, the cache index I[4:0] is generated by Xor-ing some bits of the memory address A[26:7] [4]. These Xor-ing equations can be implemented using 2-3 levels of 2-input Xor gates. It is important to note that one of the disadvantages of PRIC to be implemented in single-thread CPU cache that it exists on the critical path of the any cache access. Therefore, any cache access latency will increase by a couple of gate delays due to the latency of Xor gates [47, 150]. This latency can degrade the performance of some non-conflict cache friendly applications (i.e. applications that don't benefit form PRIC). However, when it comes to GPGPUs, the situation is different. GPGPUs are throughput-oriented architecture. Their design philosophy is built on having a large number of warps/threads per core that are interleaved with each other in order to hide long memory latency. Thus, when the cache access latency is increased by a couple of gate delays, GPGPU is able to tolerate this overhead latency (next chapter, we show that experimentally).

Comparison with Cache Bypassing Recent works [69, 98, 171] proposed cache bypassing on associativity stalls. However, these methods are not effective to efficiently utilize cache resources. In many cases, bypassing occurs while the other cache sets are underutilized. For instance, Memory Request Prioritization Buffer (MRPB) [69] allows memory request that encounters associativity stall (i.e. LINE_ALLOC_FAIL) to bypass L1 cache. As shown in figure 5.11(b), when all the 32 threads map to the same cache set, only the first four threads will successfully allocate a cache line (assume 4-way associativity) and the remaining 28 threads will bypass the L1 cache. This is because all the lines within the cache set will be reserved by the first four threads. However, our empirical search shows that the other cache sets are underutilized and thus it is better to distribute the remaining threads over the underutilized cache sets instead of bypassing. Moreover, on the second iteration, the same 32 threads access the second elements from the matrix rows and they will map to the same cache set again. The first four threads hit the cache, while

[3] For more information on the proofs and theorems behind the polynomial modulus, as well as the method to generate the H-matrix and Xor-ing equations, kindly refer to [127].

[4] In this study, We use 28-bit memory since the largest working set of our workloads is within 128 MB. We also use 128B cache line, thus the lower 7-bit A[6:0] is used to determine the byte index within the cache line and the upper 21-bit A[26:7] is used to determine the cache index and the tag address

the next four threads cause miss and consequently they evict the previous threads cache lines. This behavior is repeated over the next loop iterations, the first four threads and the second four threads evict the lines of each other. This behavior causes severe conflict misses for SYRK and GESUMMV as we have seen in Figure 4.3. On the other hand, PRIC fairly distributes memory requests over the cache sets (as shown in Figure 5.11(a)). Hence, cache bypassing is not an efficient method to handle the GPU cache associativity problem.

Comparison with High Associativity Caches Increasing the associativity of L1 cache is a straightforward approach to mitigate associativity stalls and conflict misses. Increasing the associativity of L1 cache will minimize conflict misses. Moreover, building a fully associative cache can completely eliminate all conflict misses. However, increasing associativity requires a considerable hardware overhead (tag comparators and large data selectors), which increasing both access latency and power consumption [126]. On the other hand, PRIC can acheive almost the same performance of fully asscoitive cache (as we will see in chapter 6), while incurring low hardware overhead (2-3 levels of Xor gates).

5.4 Aggregated Algorithm (DWT-PRIC)

We combine DBSA + DWT-CS + PRIC in one algorithm, named DWT-PRIC, to take advantage of all methods and achieve the highest performance. It is important to note that streaming and thrashing applications exhibit a high miss rate at L1 (as we have seen shown in figure 4.3). Thus, we need to combine DBSA and DWT-CS appropriately to avoid mischaracterize a thrashing application as streaming or vice versa. Figure 5.14 lists our algorithm DWT-PRIC and shows how to combine the three proposed techniques. First, the conventional sequential cache indexer is replaced by PRIC, such that each cache access has to pass through the PRIC xoring equations in order to avoid conflict misses and stalls. Second, to distinguish between streaming and thrashing applications, we firstly check whether it is thrashing or not. To achieve this, DWT-PRIC monitors the MPKI of L1 for the first N sampling period. If it exceeds a specific threshold for all N periods, it characterizes this application as thrashing, and DWT-CS runs an exhaustive searching to find the best number of warps. By this way, we leverage the fact that thrashing applications exhibit a high MPKI over other streaming applications (as we have seen in Figure 4.5). After DWT-CS runs for the first N sampling periods to find the appropriate number of warps that alleviates thrashing (if found), DBSA runs for the following periods till the end of the kernel. It monitors the miss rate of L1 and L2 caches. It disables L1 and/or L2 caches based on memory behavior of the running application. Once it detects a streaming behavior (i.e. miss rate exceeds a specific threshold) for a sampling period, it disables cache and bypasses memory accesses for the next sampling period. The three proposed methods are grouped together in an orthogonal and synergistic way. DWR-PRIC doesn't only cache a part of memory that contains locality, but it caches them in an efficient manner in order to avoid thrashing and conflict.

```
//DWT-PRIC
For each cache access
Cache set index = PRIC xoring equations (Memory address)

For the first N consecutive periods
If (MPKI > MPKI_threshold)  for N periods then
        Apply core sampling and find the best  number of active warps

For every sampling period till the end the of kernel
If (Missrate @L1 > Missrate_threshold)
        Disable L1 cache
Else
         Enable L1 cache

If (Missrate @L2 > Missrate_threshold)
        Disable L2 cache
Else
         Enable L2 cache
```

Figure 5.14: DWT-PRIC algorithm

5.5 Related Work

Different methods have been proposed in literature to alleviate the problems associated with GPU caches. Table 5.2 summarizes and compares between these works in terms of thrashing detection, thrashing handling and conflict stalls mitigation. We can classify these works to the following:

(1) CTA throttling:
Jog et al. [71] proposed CTA-aware-locality scheduling that gives a group of CTAs higher priority to keep their data in the L1 cache such that they get the opportunity to reuse it. Kayiran et al. [73] proposed dynamic CTA scheduling, which attempts to allocate optimal number of CTAs per-core in order to reduce contention in the memory sub-system. Lee et al. [95] explored two alternative thread block scheduling schemes. Lazy CTA scheduling was proposed to leverage GTO scheduler to determine the optimal number of CTAs per core. They also showed how block CTA scheduling (BCS), where consecutive thread blocks are assigned to the same cores, can exploit inter-block locality (i.e. intra-core and inter-core locality). It is obvious that fine-grained warp throttling mechanisms, such as DWT-CS, are better than coarse-grained CTA throttling mechanisms. Based on an experiment, as shown in Figure 5.15, static warp throttling outperforms static CTA throttling by 2X on average.

(2) Warp Throttling: In addition to the previously discussed CCWS and DAWS, other recent works were proposed to improve warp throttling. Li [98] observed that throttling techniques leave memory bandwidth and other chip resources (L2 cache, NOC and EUs) significantly underutilized. Thus, he proposed a cache bypassing scheme on top of CCWS, called Priority-based Cache Allocation (PCAL). PCAL starts from an optimal number of active warps, that alleviates thrashing and conflicts, then extra inactive warps are allowed to bypass cache and utilize the other on-chip resources. Thus, PCAL reduces

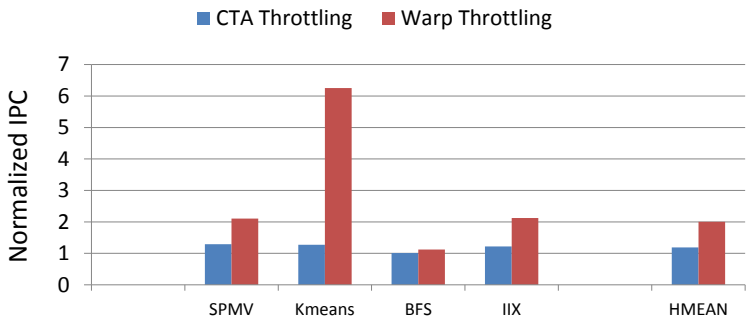

Figure 5.15: Warp throtlling vs CTA throttling performance

the cache thrashing and effectively employs the chip resources that would otherwise go unused by a pure thread throttling approach. A similar approach was proposed by Zheng et al. [171], called Adaptive Cache and Concurrency (CCA). CCA improves DAWS by allowing extra inactive warps and some streaming memory instructions from the active warps to bypass the L1 cache and utilize on-chip resources.

However, PCAL and CCA employ bypassing while leaving cache sets underutilized. For example, recall the SYRK example in section 5.3, PCAL throttles the number of active warps that can access cache to only one warp and allows two warps to bypass the cache in an attempt to utilize chip resources. However, as we have seen, the cached warp only utilizes one cache set, moreover it utilizes it in an inefficient manner (the threads map to the same set causing severe associativity stalls and conflict misses). In contrast, DWT-PRIC effectively utilizes cache sets by allowing two warps to access cache and fairly distributing their memory requests over sets. Note that, PCAL or CCA can be employed on top of our DWT-PRIC for further performance improvement and efficient utilization of L1 cache sets as well as on-chip resources.

(3) MRPB:
We have discussed MRPB in section 5.3. MRPB proposed FIFO buffers to prioritize memory requests that are generated by the same warp. It also proposed cache bypassing on associativity stalls and we have shown that the MRPB bypassing mechanism is not an effective method.

(4) Cache Replacement Policy:
Chen et al. [26] proposed G-Cache to alleviate cache thrashing. To detect thrashing, the tag array of L2 cache is enhanced with extra bits (victim bits) to provide L1 cache by some information about the hot lines that have been evicted before. An adaptive cache replacement policy is used by L1 cache to protect these hot lines. Chen [25] continued his work and proposed Coordinated Bypassing and Warp Throttling (CBWT). CBWT adopts a thrashing-resistant CPU cache management scheme, Protection Distance Prediction (PDP) [36], to GPU cache. PDP employs cache bypassing to enable protection on hot cache lines and thus alleviate cache thrashing. Excessive bypassing may over-saturate the on-chip network. Therefore, cache bypassing policy is coordinated with a dynamic warp throttling

Table 5.2: Related GPU cache contention works

	Thrashing Detection	Thrashing Handling	Conflict stalls
OWL [71]	–	CTA throttling	–
Dynamic CTA [73]	memory latency	CTA throttling	–
Lazy CTA [95]	#Instructions issued under GTO	CTA throttling	–
CCWS [132]	Victim cache	Warp throttling	–
DAWS [133]	L1 footprint prediction	Warp throttling	–
PCAL [98]	–	Warp throttling (CCWS)	Warp Bypassing
CCA [171]	L1 footprint prediction	Warp throttling (DAWS)	Warp Bypassing
MRPB [69]	–	Memory request prioritization	Bypassing on stalls
G-Cache [26]	Victims bits at L2	adaptive bypass policy	–
CBWT [25]	Victims bits at L2	PDP cache management	–
DWT+PRIC	L1 MPKI	Warp throttling (DWT-CS)	PRIC

mechanism to avoid over-saturating on-chip resources. However, the previous works don't address the associativity problem and they employ cache replacement policy to alleviate thrashing, while we use warp throttling mechanism.

Chapter 6: Experimental Results

This Chapter is organized as follows, Section 6.1 examines the performance of our proposed method over the baseline and presents a deep analysis of other benefits. Section 6.2 compares our proposed DWT-PRIC with previous works. Section 6.3 is devoted to analyze varying aspects of our design and exploring its sensitivity to design parameters.

6.1 In-depth Analysis

Figure 6.1 presents the performance improvement (Instruction Per Cycle) of our proposed methods (DWT-PRIC) on all benchmarks with respect to baseline. DWT-PRIC achieves a harmonic mean 1.54X performance improvement over baseline. It improves the average performance of streaming and contention applications by 1.2X and 2.3X respectively. Some applications exhibit an improvement up to 21X (SYRK) and 16.8X (GESUMMV). In addition, it doesn't cause any performance degradation in the cache friendly applications.

Figure 6.2 breakdowns the performance impact of each proposed technique (DBSA, DWT-CS and PRIC) and compares with the final aggregated impact of all three techniques (i.e. DWT-PRIC). DBSA, DWT-CS and PRIC improves the performance of GPGPU applications by 1.2X, 1.1X and 1.3X respectively over the baseline. When we aggregate all the techniques in one algorithm (DWT-PRIC) to take advantage of all, the performance is improved by 1.5X.

We observed that applications may benefit from one technique alone or more than one technique. Streaming applications only benefit from DBSA. Since the L1 cache is disabled in DBSA, they do not suffer from thrashing nor conflicts. PRIC and DBSA show performance improvement for Inter-warp conflict workloads, while they are insensitive to DWT-CS. However, PRIC has better performance improvement than DBSA on average. Inter-warp capacity conflict applications (thrashing-only applications) shows significant performance improvement only with DWT-CS and slight improvement with DBSA. Workloads that exhibit both Inter-warp capacity and Intra-warp conflict contention (i.e. SYRK

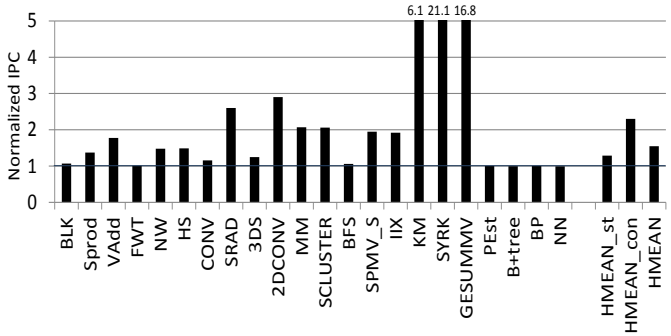

Figure 6.1: Performance improvement on all benchmarks normalized to baseline

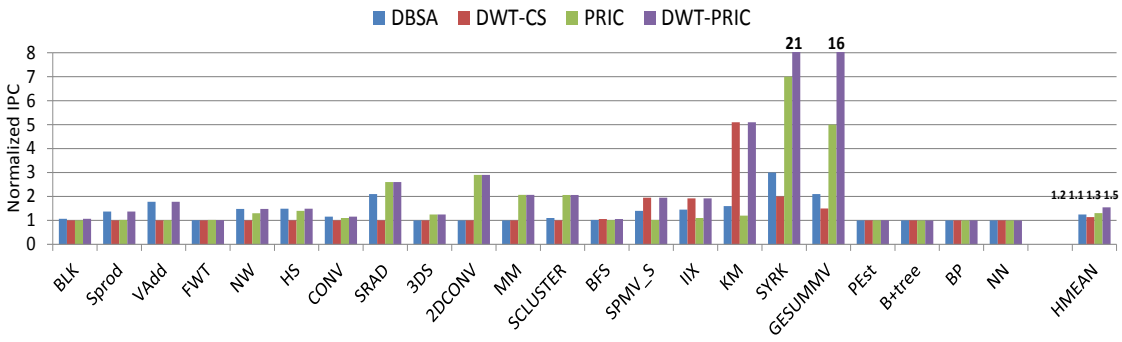

Figure 6.2: Performance improvement of each technique alone

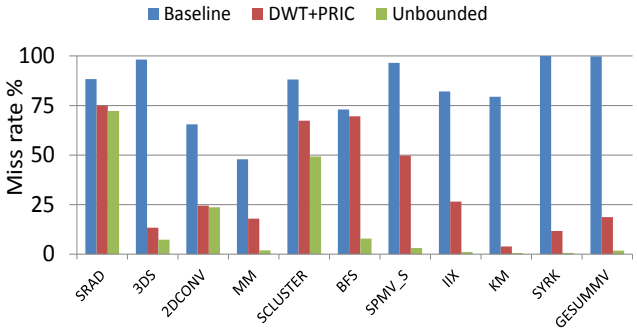

Figure 6.3: L1 miss rate reduction

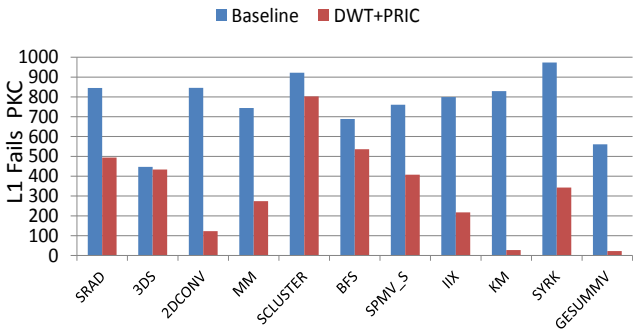

Figure 6.4: L1 reservation fails reduction

and GESUMMV) show higher performance improvement when using DBSA and PRIC, and slight improvement with DWT-CS. As shown in figure, DWT-PRIC performance is the maximum performance achieved by the the three techniques, except for (SYRK and GESUMMV). In these workloads, PRIC eliminates conflict misses and fairly distributed memory access over cache sets. Nevertheless, the cache capacity is not large enough to contain all warps working set, thus DWT-CS throttles the number of active warps to fit in cache. Therefore, these workloads significantly benefit from both DWT-CS and PRIC. Neither DWT-CS alone nor PRIC alone is able to achieve the maximum improvement for these applications.

Figure 6.3 shows the reduction in L1 miss rate for DWT-PRIC and unbounded cache compared to the baseline. More than an 80% reduction is observed under DWT-PRIC for 3DS, KMN, SYRK and GESUMMV. Also, a miss rate reduction up to 60% is noticed for 2DCONV, MM and IIX. Moreover, DWT-PRIC nearly achieves the same miss rate of unbounded cache for SRAD, 3DS, 2DCONV and KM.

Figure 6.4 shows the reduction in L1 reservation fails per kilo cycles for DWT-PRIC compared to the baseline. DWT-PRIC reduces the reservation fails by 20% on average. Applications such as 2DCONV, KM, and GESUMMV show a significant reduction (up to 90%). MM, IIX, and SYRK also show a considerable reduction (up to 60%). In contrast, SCLUSTER and SRAD still show a high level of fails PKC, especially MSHR_ENTRY_FAIL, due to the noticeable streaming behavior of these application (up to 50% and 70% miss rate in unbounded cache, as shown in Figure 6.3).

6.2 Comparison with Previous Works

We compare DWT-PRIC to previously proposed CCWS [132] and MRPB [69]. We have discussed CCWS in section 5.2. CCWS addresses the thrashing problem only and doesn't consider conflict contention. On the other hand, MRPB employs two techniques to alleviate the thrashing and conflict problems. First, a FIFO requests buffer is used to reorder memory references so that requests from the same warp are grouped and sent to the cache together and thus reducing the number of warps that access the cache at a time. Second, MRPB allows memory request that encounters associativity stall to bypass L1 cache. We have seen in section 5.3 that bypassing strategy is not effective to handle cache associativity. Table 6.1 presents the configuration parameters used for CCWS, MRPB and DWT-PRIC. In CCWS, the value Kthrottle was tuned to our baseline architecture by the same way described in [132]. In MRPB, we use the same configuration in [69] that achieved the highest performance. In DWT-PRIC, the configuration parameters were selected based on empirical analysis.

Figure 6.5 illustrates the DWT-PRIC performance improvement on cache contention applications with respect to baseline, CCWS and MRPB. In total, DWT-PRIC outperforms CCWS and MRPB by a harmonic mean 1.7X and 1.5X respectively. For inter-warp conflict contention applications, DWT-PRIC improves performance by a harmonic mean 2.1X and 1.37X over CCWS and MRPB respectively, due to the efficiency of PRIC in utilizing cache

Table 6.1: CCWS/MRPB/DWT-PRIC Configuration

CCWS Config	
Kthrottle	8
Victim Tag Array	8-way 16 entries per warp (768 total entries)
Warp Base Score	100
MRPB Config	
Signature	warp ID (resulting in 48 queues)
Drain policy	non-greedy-fixed-order
Buffer size	32 requests
Bypass option	bypass-on-assoc-stalls
DWT-PRIC Config	
Sampling Period	10K cycles
Miss rate threshold	90%
MPKI threshold	10
I-Poly	Poly(37)

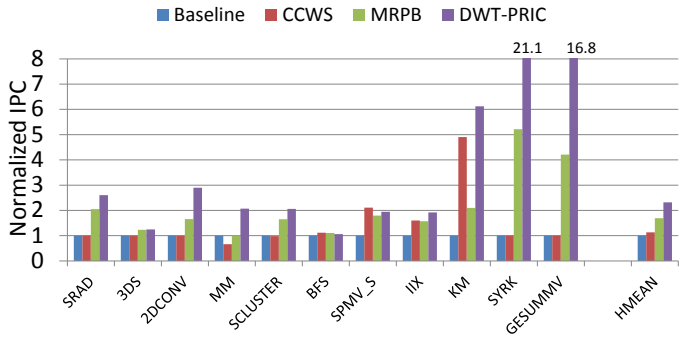

Figure 6.5: Performance improvement on contention applications compared to CCWS and MRPB

sets. For inter-warp thrashing applications, DWT-PRIC results in a harmonic mean 1.02X and 1.25X performance improvement over CCWS and MRPB respectively. DWT-PRIC shows significant improvement on the applications that show consistent thrashing level and coherent control flow divergence (IIX and KM). On the other hand, CCWS slightly performs better in SPMV and BFS due to the unsteady thrashing level of these applications. SPMV and BFS are highly control flow divergent and thus the per-warp cache footprint changes over time depending on warp's active mask. DAWS [133] is a divergence aware warp throttling mechanism that can improve the performance of these applications further. For applications that exhibit both intra-warp conflict contention and inter-warp thrashing, DWT-PRIC achieves a superior performance improvement and outperforms CCWS and MRPB by a harmonic mean 18X and 4X respectively.

Increasing the associativity of L1 cache is a straightforward approach to mitigate associativity stalls and conflict misses. For instance, recent AMD's Graphics Core Next (GCN) GPUs [5] use 64-way associativity for their 16KB L1 caches. Figure 6.6 presents PRIC

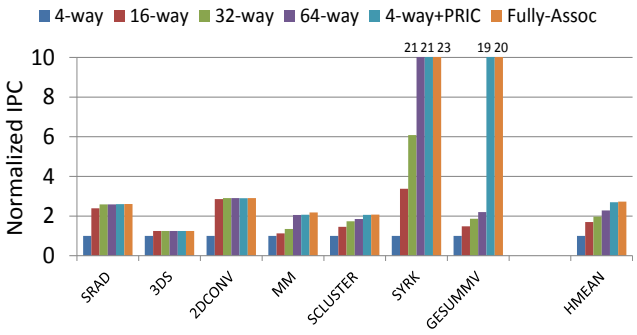

Figure 6.6: PRIC compared with high associativity

performance improvement on inter-warp and intra-warp conflict contention applications with respect to high associativity and fully-associative caches. In all cases, the L1 cache capacity is fixed and we assume idealistic 1-cycle hit latency. PRIC with 4-way associativity outperforms the 16-way, 32-way, and 64-way caches by a harmonic mean 1.6X, 1.4X, and 1.16X respectively. Moreover, PRIC is able to achieve 97% of fully-associative cache's performance. In addition to that, increasing associativity requires a considerable hardware overhead (tag comparators and large data selectors), which increases both access latency and power consumption [126].

6.3 Sensitivity Analysis

As we stated earlier in Section 5.3 that one of the limitations of PRIC is that it exists on the critical path of any cache access. However, GPGPUs rely on massive number of threads that are interleaved to hide long memory latency. Therefore, when the cache access latency is increased by one or two cycles due to employing PRIC, GPGPU is able to tolerate this overhead latency. Figure 6.7 shows pictorially PRIC latency sensitivity for our GPGPU workloads. GPGPU is able to hide latency of length one and two cycles with negligible performance loss (0.1% performance loss on average). Thrashing applications are the major cause of this performance loss. They show noticeable performance loss by 2% on average. This is because the low number of active warps (i.e. TLP) of these workloads due to employing DWT-CS throttling mechanism, and hence they lost their ability to hide latency. However, these applications still outperform the baseline architecture by an order of magnitude. Further, cache-friendly applications, that do not benefit from PRIC, are insensitive to the the increasing latency, except for Pest workload that shows 2% performance loss.

Figure 6.8 shows the sensitivity of DWT-PRIC to the L1 cache size. It depicts the performance of baseline and DWT-PRIC at various cache sizes. The results are normalized to the baseline architecture with a 16k L1 cache. The performance improvement of DWT-PRIC over the baseline with different cache size is 2.3X, 2X and 1.7X for 16KB, 32KB and 48 KB respectively. Consequently, as the cache size decreases, DWT-PRIC has a greater performance improvement relative to the baseline architecture. This is because small L1 cache size increases the occurrence of cache thrashing and conflicts due to the limited capacity and associativity.

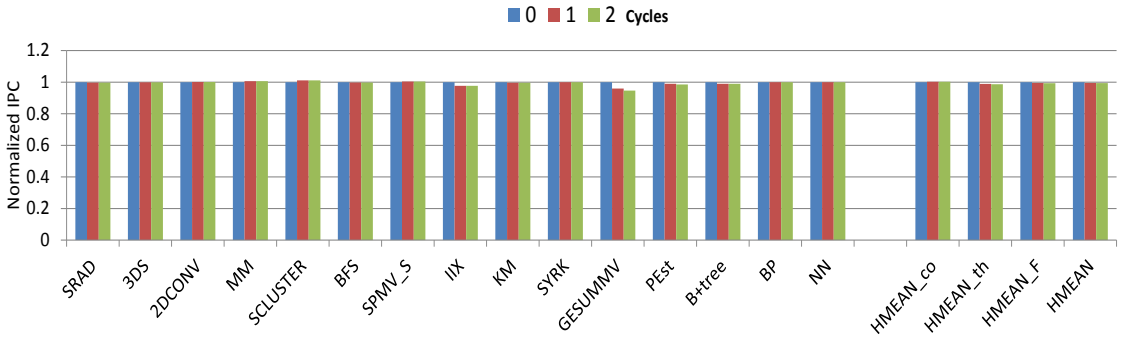

Figure 6.7: Effect of increasing PRIC latency

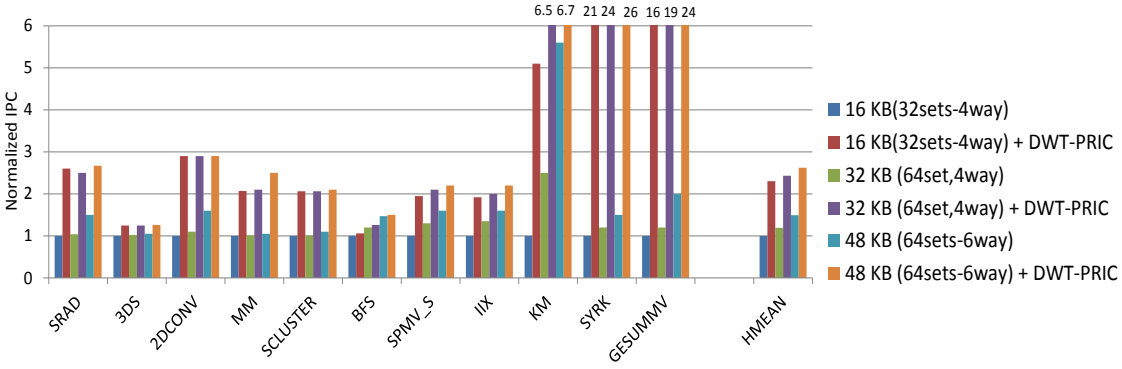

Figure 6.8: L1 cache size sensitivity

Figure 6.9 illustrates the sensitivity of PRIC to different polynomial function. We run our workloads with Poly(55) instead of Poly(37). The performance of Poly(55) was slightly different from poly(37), which implies that PRIC is is insensitive to the polynomial function.

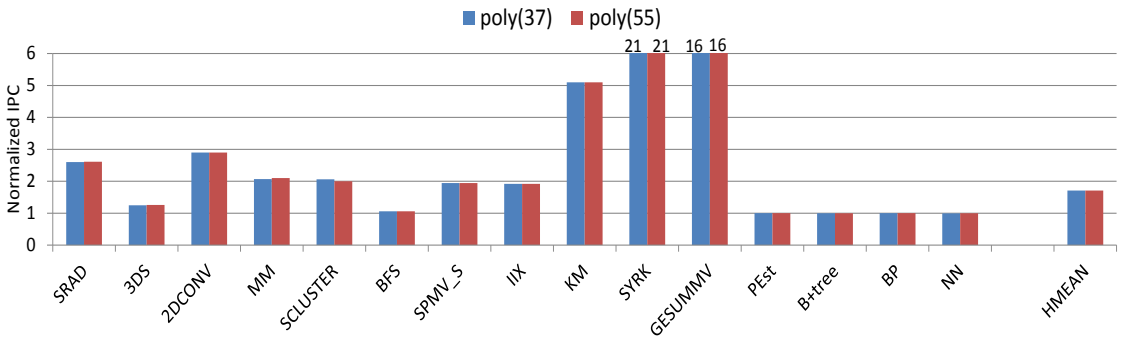

Figure 6.9: PRIC polynomial function sensitivity

Chapter 7: Conclusion and Future Work

Throughput processors, such as GPGPUs, rely on massive multithreading to hide long memory latency. However, the high number of active threads GPGPU executes concurrently leads to severe cache thrashing and conflict misses. In this work, we propose a low-cost thrashing-resistant conflict-avoiding streaming-aware GPGPU cache management scheme that efficiently utilizes the GPGPU cache resources and addresses all the problems associated with GPGPU caches. The proposed method employs three orthogonal techniques. First, it dynamically detects and bypasses streaming applications. Second, a Dynamic Warp Throttling via Cores Sampling (DWT-CS) is proposed to alleviate cache thrashing. DWT-CS runs an exhaustive searching over cores to find the best number of warps that achieves the highest performance. Third, we employ a better cache indexing function, Pseudo Random Interleaving Cache (PRIC), that is based on polynomial modulus mapping, to mitigate associativity stalls and eliminate conflict misses. Our proposed method improves the average performance of streaming and contention applications by 1.2X and 2.3X respectively. Compared to prior work, it achieves 1.7X and 1.5X performance improvement over CCWS and MRPB respectively. Moving forward, we plan to explore the following ideas:

Improving DWT-PRIC we plan to explore a more sophisticated fine-grained bypassing mechanism, instead the proposed coarse-grain cache bypassing. We also plan to improve our DWT-CS to be more resistant to unstable thrashing applications.

Fair-GPU: A Complete Fairness-aware Memory System for Concurrent GPGPU Applications Recent GPUs (NVIDIA Kepler - NVIDIA GRID technology) supports running multiple kernels from different applications context at the same GPU. In this case, the concurrent applications will share most of the GPU on-chip resources (SMs, L2 cache, network on-chip NoC and memory controller). Therefore, a fairness-aware scheme is needed to manage the concurrent applications and avoid an application to monopolize the available resources.

The GPU resources management for multitasking environments has not been well studied in literature. Jog et al. [70] proposed memory scheduling policy, first-ready round-robin FCFS (FR-RR-FCFS), that serves memory requests from different applications in a round-robin fashion while preserving DRAM page hit rates. It is important to note that, in their work, they totally omit the cache hierarchy and bypass the memory requests generated by GPU core to the memory scheduler directly. Paula et al. [3] proposed a set of different fairness policies to allocate GPU cores (i.e. SMs) to multiple concurrent applications. They also introduced a run-time approach for choosing a resource partition based on the chosen fairness policy.

During this work, we found out there are some applications that show contention at the Network on-chip (NoC), other applications show contention at both NoC and caches, while others show contention only at the main memory. To address these issues, we aim to propose a complete fairness-aware memory management scheme that provides a complete solution to manage shared resources (NoC, L2 cache and main memory) and introduce a

coordinated framework that manages fairness issues across the entire design. Ideas that were proposed to address fairness problem in CPU multicore [33, 65], will be explored and see how effective they are to address fairness problem in GPUs.

Dynamic Mixed Concurrent Application Execution Recent works [9, 95] proposed Mixed Concurrent Kernel/Application Execution, in which two applications execute concurrently on the same core. They showed how such sharing execution can improve the overall system throughput, especially mixture of memory-intensive and compute-intensive workloads. In this mixture, the compute-intensive workload's warps hide the memory latency of memory-intensive workload's warps and thus efficiently utilize the execution units. However, they just put a spotlight on the importance of this mixture of execution and its efficiency, while there are many questions that need to be answered:

1- They proposed a static approach and always mix the two running applications on the same core. However, there are some applications, when they share execution on the same core, the overall performance is degraded. To remedy this, we aim to propose a dynamic approach that initially launch two applications separately (i.e. the first application launch on the first half of cores and the second launch on the second half of cores), some statistics are collected during run-time (IPC, MPKI, shared memory and L1 cache utilization,....) and then a decision is made to mix the two applications or not. Some applications su ffer from phase execution (don't show a consistent behavior over time), thus a monitoring mechanism is required to decide whether to isolate the two applications again or not?

2- When the two applications are mixed, how the in-core resources (shared memory, L1 cache) are managed

3- Investigating a multi-application-aware warp scheduler.

Shared Resources Management for CPU+GPU Heterogeneous Architecture

The problem of shared resources management in CPU-GPU Heterogeneous architecture has been widely studied in literature. Compared with these works, we can make three new contributions:

(1) Request-Prioritization scheme:

Kayiran et al. [74] proposed GPU concurrency management that dynamically throttling/boosting TLP (i.e. number of active warps) of GPU cores in order to minimize shared resources interference between CPU and GPU. However, GPU throttling mechanism has some drawbacks. First, throttling techniques hurt some GPU applications performance that benefit from high TLP (up to 10% performance loss is observed by Kayiran et al. [74]). Second, Li et al. [98] noticed that throttling techniques may leave execution units (i.e. SIMDs) and other on-chip resources (last level cache, on-chip network and memory bandwidth) significantly underutilized. Thus, instead of throttling, we aim to propose a request-prioritization technique. In this approach, the memory request that is generated from latency-sensitive CPU applications is prioritized on its way to the main memory over the GPU memory requests. To achieve this, a prioritization-aware NoC, last level cache and memory scheduler will be proposed.

(2) GPU-application-aware resource management:

Based on our work, we observed that there are three main types of GPU workloads:

1- Graphics workloads (i.e. gaming workloads)

2- GPGPU workloads

Table 7.1: Resource management in CPU-GPU heterogeneous architecture

Resource Management	Approach	Type of GPU workloads
Network on Chip (NoC)	Virtual Channel Partitioning [92, 93]	GPGPUs
Last Level Cache	TLP-aware GPU cache bypassing [90]	GPGPUs
	TLP-aware GPU cache bypassing [105]	GPGPUs
Memory Scheduler	Staged Memory Scheduling [8]	Graphics workloads
	QoS-aware MC [66]	Graphics workloads
	cooperative heterogeneous computing memory scheduling [153]	CPU-GPGPU cooperative computing workloads
NoC + Memory Scheduler	Throttling GPU TLP [74]	GPGPUs

3- CPU-GPGPU cooperative computing workloads: The workloads in which a single parallel application is partitioned between CPU and GPU. Thus, the same application runs on CPU and GPU simultaneously. It has been shown that such workloads have their own memory characteristics that should be taken into account [153].
Each type of workloads has its own memory characteristics and QoS demands. Prior works addressed only a single type of these workloads (as shown in Table 7.1). For future work, we aim to study and analyze the behavior and memory characteristics of these different types and see how they are different form each other. Then, we aim to address all these workloads and propose a GPU-application-aware resource management.

(3) A complete fairness-aware resources management (Coordinated Resource Sharing):
Prior work has only addressed the fairness problem on one of the shared resources (Last level cache, network on chip and memory controller) individually in the absence of other resources management (as shown in table 7.1, except for [74]). However, we argue that management one of the resources will affect the other. Moving forward, we aim to propose a complete fairness-aware shared resources management and coordinate all the sharing schemes across the entire memory system to improve the overall system fairness and performance.

References

[1] Adriaens, J. T., Compton, K., Kim, N. S., and Schulte, M. J. The case for GPGPU spatial multitasking. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on* (2012), IEEE, pp. 1–12.

[2] Aguilera, P., Lee, J., Farmahini-Farahani, A., Morrow, K., Schulte, M., and Kim, N. S. Process variation-aware workload partitioning algorithms for GPUs supporting spatial-multitasking. In *Proceedings of the conference on Design, Automation & Test in Europe* (2014), p. 176.

[3] Aguilera, P., Morrow, K., and Kim, N. S. Fair Share: Allocation of GPU resources for both performance and fairness. In *Computer Design (ICCD), 2014 IEEE 32nd International Conference on* (2014), IEEE.

[4] Aguilera, P., Morrow, K., and Kim, N. S. Qos-aware dynamic resource allocation for spatial-multitasking GPUs. In *Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific* (2014).

[5] AMD. Graphics Core Next Arhcitecure whitepaper. `www.amd.com/us/Documents/GCN_Architecture_whitepaper.pdf`.

[6] AMD. AMD Fusion Kaveri. `www.amd.com/ComputeCores/`, 2014.

[7] Atoofian, E. Inter-warp prefetching for register file cache. In *12th Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD 2014)* (2014).

[8] Ausavarungnirun, R., Chang, K. K.-W., Subramanian, L., Loh, G. H., and Mutlu, O. Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems. In *Proceedings of the 39th International Symposium on Computer Architecture* (2012), IEEE Press, pp. 416–427.

[9] Awatramani, M., Zambreno, J., and Rover, D. Increasing GPU throughput using kernel interleaved thread block scheduling. In *Computer Design (ICCD), 2013 IEEE 31st International Conference on* (2013), IEEE.

[10] B Lakshminarayana, N., Lee, J., Kim, H., and Shin, J. Dram scheduling policy for GPGPU architectures based on a potential function. *Computer Architecture Letters 11*, 2 (2012), 33–36.

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

[12] Bakhoda, A., Kim, J., and Aamodt, T. M. Designing on-chip networks for throughput accelerators. *ACM Transactions on Architecture and Code Optimization TACO* (2013).

[13] Bakhoda, A., Yuan, G. L., Fung, W. W., Wong, H., and Aamodt, T. M. Analyzing CUDA workloads using a detailed GPU simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on* (2009).

[14] Blem, E., Sinclair, M., and Sankaralingam, K. Challenge benchmarks that must be conquered to sustain the GPU revolution. In *4th Workshop for Emerging Application and Many-Core Architecture (EAMA11)* (2011), vol. 1024, p. 228.

[15] Bordawekar, R., Bondhugula, U., and Rao, R. Can CPUs match GPUs on performance with productivity?: experiences with optimizing a FLOP-intensive application on CPUs and GPU. *IBM Reseach Report RC25033* (2010).

[16] Brunie, N., Collange, S., and Diamos, G. Simultaneous branch and warp interweaving for sustained GPU performance. In *Proceedings of the 39th International Symposium on Computer Architecture* (2012), IEEE Press, pp. 49–60.

[17] Cederman, D., Chatterjee, B., and Tsigas, P. Understanding the performance of concurrent data structures on graphics processors. In *Euro-Par 2012 Parallel Processing*. Springer, 2012, pp. 883–894.

[18] Cederman, D., and Tsigas, P. On dynamic load balancing on graphics processors. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware* (2008), Eurographics Association, pp. 57–64.

[19] Cederman, D., Tsigas, P., and Chaudhry, M. T. Towards a software transactional memory for graphics processors. In *Proceedings of the 10th Eurographics Conference on Parallel Graphics and Visualization* (2010), EG PGV'10, pp. 121–129.

[20] Chatterjee, N. i., O'Connor, M., Loh, G. H., Jayasena, N., and Balasubramonian, R. Managing DRAM latency divergence in irregular GPGPU applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2014), SC '14.

[21] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*.

[22] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., and Skadron, K. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of parallel and distributed computing 68*, 10 (2008), 1370–1380.

[23] Che, S., Li, J., Sheaffer, J. W., Skadron, K., and Lach, J. Accelerating compute-intensive applications with GPUs and FPGAs. In *IEEE Symposium on Application Specific Processors, 2008. SASP 2008.*, IEEE, pp. 101–107.

[24] Che, S., Sheaffer, J. W., and Skadron, K. Dymaxion: optimizing memory access patterns for heterogeneous systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (2011), ACM, p. 13.

[25] Chen, X., Chang, L.-W., Rodrigues, C. I., Ji, L., Wang, Z., and mei Hwu, W. Adaptive cache management for energy-efficient GPU computing. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (2014).

[26] Chen, X., Wu, S., Chang, L.-W., Huang, W.-S., Pearson, C., Wang, Z., and Hwu, W.-M. W. Adaptive cache bypass and insertion for many-core accelerators. In *Proceedings of International Workshop on Manycore Embedded Systems* (2014), MES '14.

[27] Choi, H., Ahn, J., and Sung, W. 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* (2012), GPGPU-5.

[28] Chung, E. S., Milder, P. A., Hoe, J. C., and Mai, K. Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs? In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture* (2010), IEEE Computer Society, pp. 225–236.

[29] Collange, S. Stack-less SIMT reconvergence at low cost. Tech. rep., Technical Report HAL-00622654, INRIA, 2011.

[30] Coutinho, B., Sampaio, D., Pereira, F. M. Q., and Meira, W. Divergence analysis and optimizations. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on* (2011), IEEE, pp. 320–329.

[31] Dally, W. J. The end of denial architecture and the rise of throughput computing. In *Keynote speech at Desgin Automation Conference* (2010).

[32] Danalis, A., Marin, G., McCurdy, C., Meredith, J. S., Roth, P. C., Spafford, K., Tipparaju, V., and Vetter, J. S. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units* (2010).

[33] Das, R., Mutlu, O., Moscibroda, T., and Das, C. Application-aware prioritization mechanisms for on-chip networks. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on* (2009).

[34] Diamond, J. R., Fussell, D. S., and Keckler, S. W. Arbitrary Modulus Indexing. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (2014).

[35] Diamos, G., Ashbaugh, B., Maiyuran, S., Kerr, A., Wu, H., and Yalamanchili, S. SIMD re-convergence at thread frontiers. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (2011), ACM, pp. 477–488.

[36] Duong, N., Zhao, D., Kim, T., Cammarota, R., Valero, M., and Veidenbaum, A. V. Improving cache management policies using dynamic reuse distances. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture* (2012), IEEE Computer Society, pp. 389–400.

[37] ElTantawy, A., Ma, J. W., OConnor, M., and Aamodt, T. M. A scalable multi-path microarchitecture for efficient GPU control flow. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on* (2014).

[38] Elteir, M., Lin, H., and Feng, W.-c. Performance characterization and optimization of atomic operations on AMD GPUs. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on* (2011), IEEE, pp. 234–243.

[39] Evegreen, A. `www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/`.

[40] Franey, S., and Lipasti, M. Accelerating atomic operations on GPGPUs. In *Networks on Chip (NoCS), 2013 Seventh IEEE/ACM International Symposium on* (2013), IEEE, pp. 1–8.

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

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

[43] Fung, W. W., Singh, I., Brownsword, A., and Aamodt, T. M. Hardware transactional memory for GPU architectures. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (2011), ACM, pp. 296–307.

[44] Garland, M., and Kirk, D. B. Understanding throughput-oriented architectures. *Communications of the ACM 53*, 11 (2010), 58–66.

[45] Gebhart, M., Johnson, D. R., Tarjan, D., Keckler, S. W., Dally, W. J., Lindholm, E., and Skadron, K. Energy-efficient mechanisms for managing thread context in throughput processors. In *ACM SIGARCH Computer Architecture News* (2011), vol. 39, ACM, pp. 235–246.

[46] Gebhart, M., Keckler, S. W., Khailany, B., Krashinsky, R., and Dally, W. J. Unifying primary cache, scratch, and register file memories in a throughput processor. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture* (2012), IEEE Computer Society, pp. 96–106.

[47] González, A., Valero, M., Topham, N., and Parcerisa, J. M. Eliminating cache conflict misses through XOR-based placement functions. In *Proceedings of the 11th international conference on Supercomputing* (1997), ACM, pp. 76–83.

[48] Grauer-Gray, S., Xu, L., Searles, R., Ayalasomayajula, S., and Cavazos, J. Auto-tuning a high-level language targeted to GPU codes. In *Innovative Parallel Computing (InPar), 2012* (2012).

[49] Gregg, C., Dorn, J., Hazelwood, K., and Skadron, K. Fine-grained resource sharing for concurrent GPGPU kernels. In *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism* (2012), USENIX Association, pp. 10–10.

[50] Gregg, C., and Hazelwood, K. Where is the data? why you cannot debate CPU vs. GPU performance without the answer. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on* (2011), IEEE, pp. 134–144.

[51] Guevara, M., Gregg, C., Hazelwood, K., and Skadron, K. Enabling task parallelism in the CUDA scheduler. In *Workshop on Programming Models for Emerging Architectures* (2009), pp. 69–76.

[52] Gwennap, L. Sandy Bridge spans generations. *Microprocessor Report 9*, 27 (2010), 10–01.

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

[54] Han, T. D., and Abdelrahman, T. S. Reducing divergence in GPGPU programs with loop merging. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units* (2013), ACM, pp. 12–23.

[55] Harper, D. T., and Jump, J. R. Vector access performance in parallel memories using a skewed storage scheme. *IEEE Transactions on Computers* (1987).

[56] He, B., Fang, W., Luo, Q., Govindaraju, N. K., and Wang, T. Mars: a MapReduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*.

[57] Hechtman, B. A., Che, S., Hower, D. R., Tian, Y., Beckmann, B. M., Hill, M. D., Reinhardt, S. K., and Wood, D. A. QuickRelease: a throughput oriented approach to release consistency on GPUs. In *Proceedings of the 20th International Symposium on High Performance Computer Architecture (HPCA)* (2014).

[58] Hechtman, B. A., and Sorin, D. J. Evaluating cache coherent shared virtual memory for heterogeneous multicore chips. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software* (2013).

[59] Hechtman, B. A., and Sorin, D. J. Exploring memory consistency for massively-threaded throughput-oriented processors. In *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)* (2013).

[60] Hensley, J. Close to the Metal. In *Proceedings of SIGGRAPH* (2007), pp. 120–130.

[61] Hill, M. D., and Smith, A. J. Evaluating associativity in CPU caches. *Computers, IEEE Transactions on* (1989).

[62] Holey, A., Mekkat, V., and Zhai, A. HAccRG: Hardware-accelerated data race detection in GPUs. In *Parallel Processing (ICPP), 2013 42nd International Conference on* (2013), IEEE, pp. 60–69.

[63] Hower, D. R., Hechtman, B. A., Beckmann, B. M., Gaster, B. R., Hill, M. D., Reinhardt, S. K., and Wood, D. A. Heterogeneous-race-free memory models. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems (APLOS)* (2014), ACM, pp. 427–440.

[64] Intel. Intel Haswell CPU. `www.intel.com/content/www/us/en/processors/core/4th-gen-core-processor-family.html`.

[65] Jaleel, A., Hasenplaugh, W., Qureshi, M., Sebot, J., Steely, Jr., S., and Emer, J. Adaptive insertion policies for managing shared caches. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (2008), PACT '08.

[66] Jeong, M. K., Erez, M., Sudanthi, C., and Paver, N. A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC. In *Proceedings of the 49th Annual Design Automation Conference* (2012), ACM, pp. 850–855.

[67] Ji, F., Lin, H., and Ma, X. RSVM: a region-based software virtual memory for GPU. In *Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on* (2013), IEEE, pp. 269–278.

[68] Jia, W., Shaw, K. A., and Martonosi, M. Characterizing and improving the use of demand-fetched caches in GPUs. In *Proceedings of the 26th ACM international conference on Supercomputing* (2012), ACM, pp. 15–24.

[69] Jia, W., Shaw, K. A., and Martonosi, M. a. MRPB: Memory request prioritization for massively parallel processors. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on* (2014).

[70] Jog, A., Bolotin, E., Guz, Z., Parker, M., Keckler, S. W., Kandemir, M. T., and Das, C. R. Application-aware memory system for fair and efficient execution of concurrent GPGPU applications. In *Proceedings of Workshop on General Purpose Processing Using GPUs* (2014), ACM, p. 1.

[71] Jog, A., Kayiran, O., Chidambaram Nachiappan, N., Mishra, A. K., Kandemir, M. T., Mutlu, O., Iyer, R., and Das, C. R. 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)* (2013), ACM, pp. 395–406.

[72] Jooybar, H., Fung, W. W., O'Connor, M., Devietti, J., and Aamodt, T. M. GPUDet: a deterministic GPU architecture. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems (ASPLOS)* (2013), ACM, pp. 1–12.

[73] Kayiran, O., Jog, A., Kandemir, M. T., and Das, C. R. Neither more nor less: Optimizing thread-level parallelism for GPGPUs. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques* (2013), IEEE Press, pp. 157–166.

[74] Kayiran, O., Nachiappan, N. C., Jog, A., Ausavarungnirun, R., Kandemir, M. T., Loh, G. H., Mutlu, O., and Das, C. R. Managing GPU concurrency in heterogeneous architectures. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (2014).

[75] Keckler, S. W., Dally, W. J., Khailany, B., Garland, M., and Glasco, D. GPUs and the future of parallel computing. *Micro, IEEE 31*, 5 (2011), 7–17.

[76] Kharbutli, M., Irwin, K., Solihin, Y., and Lee, J. Using prime numbers for cache indexing to eliminate conflict misses. In *Software, IEEE Proceedings-* (2004).

[77] KHRONOS Group. The OpenCL Specification version 2.0. `https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf`.

[78] Kilgariff, E., and Fernando, R. The GeForce 6 series GPU architecture. In *ACM SIGGRAPH 2005 Courses* (2005), ACM, p. 29.

[79] Kim, G., Lee, M., Jeong, J., and Kim, J. Multi-GPU system design with memory network. In *Proceedings of the 2010 47th Annual IEEE/ACM international symposium on Microarchitecture* (2014).

[80] Kim, H. Supporting virtual memory in GPGPU without supporting precise exceptions. In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness* (2012), ACM, pp. 70–71.

[81] Kim, Y., Lee, H., and Kim, J. An alternative memory access scheduling in manycore accelerators. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on* (2011), IEEE, pp. 195–196.

[82] Kim, Y., Lee, J., Jo, J.-E., and Kim, J. GPUdmm: A high-performance and memory-oblivious GPU architecture using dynamic memory management. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on* (2014).

[83] Kim, Y., Lee, J., Kim, D., and Kim, J. ScaleGPU: GPU architecture for memory-unaware GPU programming. *Computer Architecture Letters PP*, 99 (2013), 1–1.

[84] Kirk, D., and Wen-mei, W. *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann, 2010.

[85] Kyriazis, G. Heterogeneous system architecture: A technical review. `http://developer.amd.com/wordpress/media/2012/10/hsa10.pdf`, 2012.

[86] Lakshminarayana, N. B., and Kim, H. Effect of instruction fetch and memory scheduling on GPU performance. In *Workshop on Language, Compiler, and Architecture Support for GPGPU* (2010).

[87] Lakshminarayana, N. B., and Kim, H. Spare register aware prefetching for graph algorithms on GPUs. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on* (2014).

[88] Lashgar, A., Khonsari, A., and Baniasadi, A. HARP: Harnessing inactive threads in many-core processors. *ACM Transactions on Embedded Computing Systems (TECS) 13*, 3s (2014), 114.

[89] Lawrie, D. H., and Vora, C. R. The prime memory system for array access. *IEEE transactions on Computers* (1982).

[90] Lee, J., and Kim, H. TAP: A TLP-aware cache management policy for a CPU-GPU heterogeneous architecture. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on* (2012), IEEE, pp. 1–12.

[91] Lee, J., Lakshminarayana, N. B., Kim, H., and Vuduc, R. Many-thread aware prefetching mechanisms for GPGPU applications. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on* (2010), IEEE, pp. 213–224.

[92] Lee, J., Li, S., Kim, H., and Yalamanchili, S. Adaptive virtual channel partitioning for network-on-chip in heterogeneous architectures. *ACM Transactions on Design Automation of Electronic Systems (TODAES) 18*, 4 (2013), 48.

[93] Lee, J., Li, S., Kim, H., and Yalamanchili, S. Design space exploration of on-chip ring interconnection for a CPU-GPU heterogeneous architecture. *J. Parallel Distrib. Comput. 73*, 12 (2013), 1525–1538.

[94] Lee, J., Samadi, M., Park, Y., and Mahlke, S. Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques* (2013), IEEE Press, pp. 245–256.

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

[96] Lee, Y., Grover, V., Krashinsky, R., Stephenson, M., Keckler, S. W., and Asanovic, K. Exploring the design space of SPMD divergence management on data-parallel architectures. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (2014).

[97] Lee, Y., Krashinsky, R., Grover, V., Keckler, S. W., and Asanovic, K. Convergence and scalarization for data-parallel architectures. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on* (2013), IEEE, pp. 1–11.

[98] Li, D. *Orchestrating Thread Scheduling and Cache Management to Improve Memory System Throughput in Throughput Processors*. PhD thesis, The University Of Texas At Austin, May 2014.

[99] Lin, Y., Tang, T., and Wang, G. Power optimization for GPU programs based on software prefetching. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on* (2011).

[100] Lindholm, E., Kilgard, M. J., and Moreton, H. A user-programmable vertex engine. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2001), ACM, pp. 149–158.

[101] Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J. NVIDIA Tesla: A unified graphics and computing architecture. *Micro, IEEE* (2008).

[102] Liu, W. Y. B. W. Z., and Wang, Y. An initial study on density-aware cache management for GPU.

[103] Lustig, D., and Martonosi, M. Reducing GPU offload latency via fine-grained CPU-GPU synchronization. In *High Performance Computer Architecture (HPCA), 2013 IEEE 19th International Symposium on* (Feb 2013), pp. 354–365.

[104] Mathworld. `mathworld.wolfram.com/IrreduciblePolynomial.html`.

[105] Mekkat, V., Holey, A., Yew, P.-C., and Zhai, A. Managing shared last-level cache in a heterogeneous multicore processor. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques (PACT)* (2013), IEEE Press, pp. 225–234.

[106] Meltzer, R., Zeng, C., and Cecka, C. Micro-benchmarking the C2070. In *GPU Technology Conference* (2013), Citeseer.

[107] Meng, J., and Skadron, K. Avoiding cache thrashing due to private data placement in last-level cache for manycore scaling. In *Computer Design, 2009. ICCD 2009. IEEE International Conference on* (2009), IEEE, pp. 282–288.

[108] Meng, J., Tarjan, D., and Skadron, K. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (2010), ISCA '10.

[109] Menon, J., De Kruijf, M., and Sankaralingam, K. iGPU: exception support and speculative execution on GPUs. In *Proceedings of the 39th International Symposium on Computer Architecture (ISCA)* (2012), IEEE Press, pp. 72–83.

[110] Misra, P., and Chaudhuri, M. Performance evaluation of concurrent lock-free data structures on GPUs. In *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on* (2012), IEEE, pp. 53–60.

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

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

[113] NVIDIA. CUDA C/C++ SDK Code Samples. `http://developer.nvidia.com/cuda-cc-sdk-code-samples`.

[114] NVIDIA. NVIDIAs Next Generation CUDA Compute Architecture: Kepler GK110. nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[115] NVIDIA. Project Denver. http://blogs.nvidia.com/blog/2011/01/05/project-denver-processor-to-usher-in-new-era-of-computing/.

[116] NVIDIA. CUDA C Programming guide v6.5. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, 2015.

[117] Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C. GPU computing. *Proceedings of the IEEE 96*, 5 (2008), 879–899.

[118] Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., and Purcell, T. J. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum* (2007), vol. 26, Wiley Online Library, pp. 80–113.

[119] Pai, S., Govindarajan, R., and Thazhuthaveetil, M. J. Fast and efficient automatic memory management for GPUs using compiler-assisted runtime coherence scheme. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques* (2012), ACM, pp. 33–42.

[120] Pai, S., Thazhuthaveetil, M. J., and Govindarajan, R. Improving GPGPU concurrency with elastic kernels. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems* (2013), ACM, pp. 407–418.

[121] Paulius Micikevicius. GPU performance analysis and optimization. http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0514-GTC2012-GPU-Performance-Analysis.pdf, 2012.

[122] Pei, S., Kim, M.-S., Gaudiot, J.-L., and Xiong, N. Fusion coherence: Scalable cache coherence for heterogeneous Kilo-Core system. In *Advanced Computer Architecture*. Springer, 2014.

[123] Pichai, B., Hsu, L., and Bhattacharjee, A. Architectural support for address translation on GPUs: designing memory management units for CPU/GPUs with unified address spaces. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems* (2014), ACM, pp. 743–758.

[124] Power, J., Basu, A., Gu, J., Puthoor, S., Beckmann, B. M., Hill, M. D., Reinhardt, S. K., and Wood, D. A. Heterogeneous system coherence for integrated CPU-GPU systems. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (2013), ACM, pp. 457–467.

[125] Power, J., Hill, M., and Wood, D. Supporting x86-64 address translation for 100s of GPU lanes. In *Proceedings of the 20th International Symposium on High Performance Computer Architecture* (2014).

[126] Qureshi, M. K., Thompson, D., and Patt, Y. N. The V-Way cache: demand-based associativity via global replacement. In *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on* (2005).

[127] Rau, B. R. Pseudo-randomly interleaved memory. In *Proceedings of the 18th Annual International Symposium on Computer Architecture* (1991), ISCA '91.

[128] Rhu, M., and Erez, M. CAPRI: prediction of compaction-adequacy for handling control-divergence in GPGPU architectures. In *Proceedings of the 39th International Symposium on Computer Architecture* (2012), IEEE Press, pp. 61–71.

[129] Rhu, M., and Erez, M. The dual-path execution model for efficient GPU control flow. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture* (2013), HPCA '13, pp. 591–602.

[130] Rhu, M., and Erez, M. Maximizing simd resource utilization in GPGPUs with SIMD lane permutation. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (2013), ISCA '13.

[131] Rhu, M., Sullivan, M., Leng, J., and Erez, M. A locality-aware memory hierarchy for energy-efficient GPU architectures. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (2013), ACM, pp. 86–98.

[132] Rogers, T. G., O'Connor, M., and Aamodt, T. M. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture* (2012), IEEE Computer Society, pp. 72–83.

[133] Rogers, T. G., O'Connor, M., and Aamodt, T. M. Divergence-aware warp scheduling. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (2013), ACM, pp. 99–110.

[134] Samadi, M., Hormati, A., Lee, J., and Mahlke, S. Paragon: collaborative speculative loop execution on GPU and CPU. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units* (2012), ACM, pp. 64–73.

[135] Samadi, M., Lee, J., Jamshidi, D. A., Hormati, A., and Mahlke, S. SAGE: self-tuning approximation for graphics engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (2013), ACM, pp. 13–24.

[136] Sartori, J., and Kumar, R. Branch and data herding: Reducing control and memory divergence for error-tolerant GPU applications. *IEEE Transactions on Multimedia 15*, 2 (2013), 279–290.

[137] Sathish, V., Schulte, M. J., and Kim, N. S. Lossless and lossy memory I/O link compression for improving performance of GPGPU workloads. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques* (2012), ACM, pp. 325–334.

[138] Sethia, A., Dasika, G., Samadi, M., and Mahlke, S. APOGEE: adaptive prefetching on GPUs for energy efficiency. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques* (2013), IEEE Press, pp. 73–82.

[139] Seznec, A. A case for two-way skewed-associative caches. In *ACM SIGARCH Computer Architecture News* (1993), ACM, pp. 169–178.

[140] Singh, I., Shriraman, A., Fung, W. W., O'Connor, M., and Aamodt, T. M. Cache coherence for GPU architectures. In *Proceedings of the 20th International Symposium on High Performance Computer Architecture* (2013), HPCA '13, pp. 578–590.

[141] Sohi, G. S. Logical data skewing schemes for interleaved memories in vector processors. *Technical Report, University of Wisconsin Computer Sciences Dept., Madison, WI* (1988).

[142] Song, S., Lee, M., Kim, J., Seo, W., Cho, Y., and Ryu, S. Energy-efficient scheduling for memory-intensive GPGPU workloads. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014* (2014), IEEE, pp. 1–6.

[143] Sorensen, T., Gopalakrishnan, G., and Grover, V. Towards shared memory consistency models for GPUs. In *Proceedings of the 27th international ACM conference on International conference on supercomputing* (2013), ACM, pp. 489–490.

[144] Steffen, M., and Zambreno, J. Improving SIMT efficiency of global rendering algorithms with architectural support for dynamic micro-kernels. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on* (2010), IEEE, pp. 237–248.

[145] Stuart, J. A., and Owens, J. D. Efficient synchronization primitives for GPUs. *arXiv preprint arXiv:1110.4623* (2011).

[146] Tanasic, I., Gelado, I., Cabezas, J., Ramirez, A., Navarro, N., and Valero, M. Enabling preemptive multiprogramming on GPUs. In *Proceedings of the 41th International Symposium on Computer Architecture (ISCA)* (2014).

[147] Tarjan, D., Meng, J., and Skadron, K. Increasing memory miss tolerance for SIMD cores. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (2009), ACM, p. 22.

[148] Tarjan, D., and Skadron, K. 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* (2010), IEEE Computer Society, pp. 1–10.

[149] Thompson, C. J., Hahn, S., and Oskin, M. Using modern graphics architectures for general-purpose computing: a framework and analysis. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture* (2002), IEEE Computer Society Press, pp. 306–317.

[150] Topham, N., González, A., and González, J. The design and performance of a conflict-avoiding cache. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture* (1997).

[151] Vaidya, A. S., Shayesteh, A., Woo, D. H., Saharoy, R., and Azimi, M. SIMD divergence optimization through intra-warp compaction. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (2013), ACM, pp. 368–379.

[152] Wald, I. Active thread compaction for GPU path tracing. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics* (2011), ACM, pp. 51–58.

[153] Wang, H., Singh, R., Schulte, M. J., and Kim, N. S. Memory scheduling towards high-throughput cooperative heterogeneous computing. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation* (2014), PACT '14.

[154] Wang, L., Huang, M., and El-Ghazawi, T. Exploiting concurrent kernel execution on graphic processing units. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on* (2011), IEEE, pp. 24–32.

[155] Wang, Y., Chen, S., Wan, J., Meng, J., Zhang, K., Liu, W., and Ning, X. A multiple SIMD, multiple data (MSMD) architecture: Parallel execution of dynamic and static SIMD fragments. In *Proceedings of the 20th International Symposium on High Performance Computer Architecture* (2013), HPCA '13, pp. 603–614.

[156] Wittenbrink, C. M., Kilgariff, E., and Prabhu, A. Fermi GF100 GPU architecture. *Micro, IEEE* (2011).

[157] Woo, D. H., and Lee, H.-H. S. COMPASS: a programmable data prefetcher using idle GPU shaders. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*(2010).

[158] Wu, B., Zhao, Z., Zhang, E. Z., Jiang, Y., and Shen, X. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on GPU. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming* (2013), ACM, pp. 57–68.

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

[160] Xiao, S., and Feng, W.-c. Inter-block GPU communication via fast barrier synchronization. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on* (2010), IEEE, pp. 1–12.

[161] Xie, X., Liang, Y., Sun, G., and Chen, D. An efficient compiler framework for cache bypassing on GPUs. In *Computer-Aided Design (ICCAD), 2013 IEEE/ACM International Conference on* (2013).

[162] Xu, Y., Wang, R., Goswami, N., Li, T., Gao, L., and Qian, D. Software transactional memory for GPU architectures. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (2014), ACM, p. 1.

[163] Yang, Y., Xiang, P., Mantor, M., Rubin, N., and Zhou, H. Shared memory multiplexing: A novel way to improve GPGPU performance. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques* (2012), PACT '12.

[164] Yang, Y., Xiang, P., Mantor, M., and Zhou, H. CPU-assisted GPGPU on fused CPU-GPU architectures. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on* (2012), IEEE, pp. 1–12.

[165] Yilmazer, A., and Kaeli, D. HQL: A scalable synchronization mechanism for GPUs. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on* (2013), IEEE, pp. 475–486.

[166] Yu, Y., He, X., Guo, H., Wang, Y., and Chen, X. A credit-based load-balance-aware CTA scheduling optimization scheme in GPGPU. *International Journal of Parallel Programming* (2014), 1–21.

[167] Yuan, G. L., Bakhoda, A., and Aamodt, T. M. Complexity effective memory access scheduling for many-core accelerator architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (2009), ACM, pp. 34–44.

[168] Zhang, E. Z., Jiang, Y., Guo, Z., Tian, K., and Shen, X. On-the-fly elimination of dynamic irregularities for GPU computing. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2011), ACM.

[169] Zheng, M., Ravi, V. T., Qin, F., and Agrawal, G. GRace: a low-overhead mechanism for detecting data races in GPU programs. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11.

[170] Zheng, M., Ravi, V. T., Qin, F., and Agrawal, G. Gmrace: Detecting data races in GPU programs via a low-overhead scheme. *IEEE Transactions on Parallel and Distributed Systems 25*, 1 (2014), 104–115.

[171] Zheng, Z., Wang, Z., and Lipasti, M. Adaptive cache and concurrency allocation on GPGPUs. *Computer Architecture Letters PP*, 99 (2014).

[172] Zhong, J., and He, B. Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling. *IEEE Transactions on Parallel and Distributed Systems* (2013).

الملخص

وحدات معالجات الرسومات الحديثة مجهزة بمستويين من الذاكرة المؤقتة للأستخدام العام فى محاولة للحد من المتطلبات العالية للقدرات الانتاجية للذاكرة و تحسين اداء بعض التطبيقات الغير منتظمة. ولكن, معالجات عالية الانتاج, مثل وحدات معالجة الرسومات, تعتمد على تناوب عدد هائل من العمليات مع بعضها البعض و ذلك من أجل اخفاء الوقت الطويل الذى يستخدم لقراءة الذاكرة الرئيسية. ولكن, هذا العدد الهائل من العمليات التى تنفذ فى نفس الوقت يؤدى الى اختناق و ازدحام الذاكرة المؤقته الذى من الممكن أن يؤدى الى نتائج عكسية و اضعاف الاداء.

يقدم هذا البحث طريقه جديدة منخفضة التكلفة تعمل على التخفيف من الاختناق و معالجة الازدحام الحادث فى الذاكرة المؤقتة لوحدات معالجة الرسومات. تقوم الطريقة المقترحة على ثلاثة أفكار رئيسية. أولا, اتاحة تجاوز او اغلاق الذاكرة المؤقتة عندما تكون الذاكرة عديمة الفائدة وذلك اثناء تنفيذ التطبيقات المتدفقة. ثانيا, يقدم البحث طريقة جديدة لمعالجة اختناق الذاكرة المؤقتة و ذلك عن طريق التقليل من عدد العمليات النشطة. هذه الطريقة الجديدة تعتمد على أخذ عينات من المعالجات بشكل ديناميكى لايجاد افضل عدد من العمليات التى يمكن ان تنفذ فى نفس الوقت دون احداث ازدحام فى الذاكره المؤقتة. ثالثا, يستخدم البحث طريقة أفضل, تسمى التداخل العشوائى, لتوزيع الطلبات على مجموعات الذاكرة المؤقته. تعتمد هذة الطريقة على استخدام معادلة معامل متعدد الحدود بدلا من الطريقة المتعاقبة التقليدية وهذا يؤدى الى توزيع الطلبات بشكل متساوى و عادل على مجموعات الذاكرة المؤقتة. الطريقة المقترحة تحسن من اداء التطبيقات المتدفقة و المزدحمة بمعدل ١.٢ و ٢.٣ على التوالى. مقارنة بالطرق المقترحة السابقة, جدولة العمليات الواعى للذاكرة للمؤقتة و ترتيب الاوليات لطلبات الذاكرة المؤقتة, فهو يحسن الاداء بمعدل ١.٧ و ١.٥ على التوالى.

مهندس: محمود خيرى عبدالصادق عبدالله
تاريخ الميلاد: ١٣ \ ٦ \ ١٩٨٩
الجنسية: مصرى
تاريخ التسجيل: ١٥ \ ٩ \ ٢٠١١
تاريخ المنح: ١ \ ١ \ ٢٠١٥
القسم: هندسة الحاسبات
الدرجة: ماجستير العلوم هندسة الحاسبات
المشرفون:
أ.م.د. عمرو جلال وصال
الممتحنون:
أ.د. محمد واثق على كامل الخراشى (الممتحن الخارجي)
أ.م.د. حسام على حسن فهمى (الممتحن الداخلي)
أ.م.د. عمرو جلال الدين وصال (المشرف الرئيسي)

عنوان الرسالة:

الاستخدام الكفء للذاكرة المرحلية لوحدات معالجة الرسومات

الكلمات الدالة:
إدارة الذاكرة المؤقتة، الاستخدام العام لوحدات معالج الرسومات، خنق العمليات، تجنب النزاع، تجاوز الذاكرة المؤقتة

ملخص الرسالة:
معالجات عالية الانتاج, مثل وحدات معالجة الرسومات, تعتمد على تناوب عدد هائل من العمليات مع بعضها البعض و ذلك من أجل اخفاء الوقت الطويل التى يستخدم لقراءة الذاكرة الرئيسية. ولكن, هذا العدد الهائل من العمليات التى تنفذ فى نفس الوقت يؤدى الى اختناق و ازدحام الذاكرة المؤقته. يقدم هذا البحث طريقه جديدة منخفضة التكلفة تعمل على التخفيف من الاختناق و معالجة الازدحام الحادث فى الذاكرة المؤقتة لوحدات معالجة الرسومات. تقوم الطريقة المقترحة على ثلاثة أفكار رئيسة. أولا, اتاحة تجاوز او اغلاق الذاكرة المؤقتة عندما تكون الذاكرة عديمة الفائدة وذلك اثناء تنفيذ التطبيقات المتدفقة. ثانيا, يقدم البحث طريقة جديدة لمعالجة اختناق الذاكرة المؤقتة و ذلك عن طريق التقليل من عدد العمليات النشطة. هذه الطريقة الجديدة تعتمد على اخذ عينات من المعالجات بشكل ديناميكى لايجاد افضل عدد من العمليات التى يمكن ان تنفذ فى نفس الوقت دون احداث ازدحام فى الذاكره المؤقتة. ثالثا, يستخدم البحث طريقة أفضل, تسمى التداخل العشوائى, لتوزيع الطلبات على مجموعات الذاكرة المؤقته بشكل متساوى و عادل. الطريقة المقترحة تحسن من اداء التطبيقات المتدفقة و المزدحمة بمعدل ١.٢ و ٢.٣ على التوالى.

الاستخدام الكفء للذاكرة المرحلية لوحدات معالجة الرسومات

اعداد
محمود خيرى عبدالصادق عبدالله

رسالة مقدمة إلى كلية الهندسة - جامعة القاهرة
كجزء من متطلبات الحصول على درجة ماجستير العلوم
في
هندسة الحاسبات

يعتمد من لجنة الممتحنين:

الدكتور: عمرو جلال الدين وصال المشرف الرئيسى

الدكتور: حسام على حسن فهمى الممتحن الداخلي

الاستاذ الدكتور: محمد واثق على كامل الخراشى الممتحن الخارجي
أستاذ بقسم هندسة الحاسبات و النظم - كلية الهندسة - جامعة عين شمس

كليــة الهندســة - جامعــة القاهــرة
الجيـزة - جمهوريـة مصـر العربيــة

٢٠١٥

الاستخدام الكفء للذاكرة المرحلية لوحدات معالجة الرسومات

اعداد
محمود خيرى عبدالصادق عبدالله

رسالة مقدمة إلى كلية الهندسة - جامعة القاهرة
كجزء من متطلبات الحصول على درجة ماجستير العلوم
في
هندسة الحاسبات

تحت اشراف

أ.م.د./ عمرو جلال الدين وصال
أستاذ مساعد بقسم هندسة الحاسبات
كليــة الهندســة - جامعــة القاهــرة

كليــة الهندســة - جامعــة القاهــرة
الجيـزة - جمهوريـة مصـر العربيــة

٢٠١٥

الاستخدام الكفء للذاكرة المرحلية لوحدات معالجة الرسومات

اعداد

محمود خيرى عبدالصادق عبدالله

رسالة مقدمة إلى كلية الهندسة - جامعة القاهرة
كجزء من متطلبات الحصول على درجة ماجستير العلوم
في
هندسة الحاسبات

كليــة الهندســة - جامعــة القاهــرة
الجيـزة - جمهوريـة مصـر العربيــة

٢٠١٥