Mitigating GPU Memory Divergence for Data-Intensive Applications

by

Bin Wang

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
August 1, 2015

Keywords: GPU, Memory Divergence, Intra-Warp Conflicts, Cache Indexing Method, Memory Occlusion, Warp Scheduling

Approved by

Weikuan Yu, Chair, Associate Professor of Computer Science and Software Engineering
Sanjeev Baskiyar, Associate Professor of Computer Science and Software Engineering
Soo-Young Lee, Professor of Electrical and Computer Engineering

Abstract

Graphics Processing Units (GPUs) have proven as a viable technology for a wide variety of general purpose applications to exploit the massive computing capability and high computation efficiency. In GPUs, threads are organized into warps and threads in a warp execute in lock-step. GPUs deliver massive parallelism by alternating the execution of many concurrent warps and overlapping the long latency off-chip memory accesses of some warps with the computation of other warps. Following the success of GPU accelerations for compute-intensive high performance computing applications, the arrival of big data era has energized a new trend of GPU accelerations for data-intensive applications. Pioneering works have demonstrated that GPU-based implementations of data-intensive applications can provide significant performance improvement over traditional CPU-based implementations.

However, due to the complexity in managing GPU on-chip resources through high level programming languages and the complicated memory access patterns in data-intensive applications, it often takes tremendous efforts to optimize these applications for high performance. Memory divergence is a major performance bottleneck that prevents data-intensive applications from gaining high performance in GPUs. In the lock-step execution model, memory divergence refers to the case where intra-warp accesses cannot be coalesced into one or two cache blocks. Even though the impacts of memory divergence can be alleviated through various software techniques, architectural support for memory divergence mitigation is still highly desirable to ease the complexity in the programming and optimization of GPU-accelerated data-intensive applications.

When memory divergence occurs, a warp incurs up to warp-size (e.g., 32) independent cache accesses. Such a burst of divergent accesses not only generates large volume of long latency off-chip memory operations, but also exhibits three new architectural challenges, including *intra-warp associativity conflicts*, *partial caching*, and *memory occlusion*. To be more specific, intra-warp

associativity conflicts are caused by the pathological behaviors of current cache indexing method that concentrates divergent intra-warp memory accesses into a few cache sets. Divergent memory accesses are often associated with high intra-warp locality, but current cache management can not manage all cache lines of a warp as a unit, leading to severe partial caching of high intra-warp locality. Memory occlusion is a structural hazard in the GPU pipeline that occurs when the available Memory Status History Register (MSHR) entries are insufficient to track all the memory requests of a divergent load. In current GPUs, replaying missing memory accesses that are caused by associativity conflicts, intra-warp locality loss, and MSHR unavailability is a common approach to overcome the three challenges. However, replaying memory accesses stalls the execution in Load/Store (LD/ST) units and eventually impacts the instruction throughput in warp schedulers, severely degrading the performance of executing memory divergent benchmarks on GPUs.

This dissertation introduces three novel and light-weight architectural modifications to independently solve the three challenges: 1) a *Full Permutation (FUP)* based GPU cache indexing method is presented to uniformly disperse intra-warp accesses into all available cache sets so that associativity conflicts can be eliminated; 2) a *Divergence-Aware Cache (DaCache)* Management technique is designed to orchestrate warp scheduling and cache management, make caching decisions at the granularity of individual warps, reduce partial caching of high intra-warp locality, and resist inter- and intra-warp cache thrashing; and 3) a *Memory Occlusion Aware Warp Scheduler (OAWS)* is proposed to dynamically predict the MSHR consumption of each divergent load instruction and only schedule warps that will not incur memory occlusion.

The proposed techniques are implemented in a cycle-accurate GPGPU simulator, and compared with closely related state-of-the-art techniques. Specifically, *FUP* is compared with conventional indexing method, Bitwise-XOR, and Prime Number Displacement; *DaCache* is compared with two representative thrashing resistant cache management techniques, Dynamic Insertion Policy (DIP) and Re-Reference Interval Prediction (RRIP); and *OAWS* is compared with state-of-the-art warp scheduling techniques that can mitigate the impacts of memory occlusion, including Static Warp Limiting (SWL), Cache Conscious Wavefront Scheduling (CCWS), and Memory Aware

Scheduling and Cache Access Re-execution (MASCAR). Data-intensive workloads from various publically available GPU benchmark suites are used for performance evaluations. Through systematic experiments and comprehensive comparisons with existing state-of-the-art techniques, this dissertation has demonstrated the effectiveness of our aforementioned techniques and the viability of mitigating memory divergence through architectural support. Meanwhile, this dissertation reveals optimization spaces for proposed solutions and other promising opportunities for future research on GPU architecture.

keywords: *GPU, Memory Divergence, Intra-Warp Conflicts, Cache Indexing Method, Memory Occlusion, Warp Scheduling*

Acknowledgments

The completion of the dissertation would not have been possible without the help from my advisor, my committee members, my reader, my lab mates, and my family.

Foremost, I am grateful for my advisor, Dr. Weikuan Yu, for his consistent support, encouragement, and supervision during my Ph.D studies. Without his generous financial support, I would not have had the opportunity to study in the U.S. and pursue my PhD at Auburn University. Dr. Yu has always unconditionally given me advice and patiently refined my research ideas. His attitudes towards good research and work have set a high bar for me to build my research portfolio. His rigorous research spirit will benefit me and my career in the rest of my life.

Furthermore, I would like to thank my committee members, Dr. Sanjeev Baskiyar, and Dr. Soo-Young Lee, and the outside reader of this dissertation, Dr. Shiwen Mao. They have given me valuable advice on my dissertation.

I would also like to acknowledge the generous help I have received from members of The Parallel Architecture and System Research Lab (PASL) at Auburn University. I will forever appreciate the friendship with Dr. Yuan Tian, Dr. Xinyu Que, Dr. Yandong Wang, Dr. Cong Xu, Dr. Zhuo Liu, Dr. Cristi Cira, Patrick Carpenter, Dr. Chunxiang Wu, Xuechao Li, Xiaobing Li, Teng Wang, Yizheng Jiao, Fang Zhou, Huansong Fu, Xinning Wang, Kevin Vasko, Michael Pritchard, Dr. Hui Chen, Dr. Jianhui Yue, Yue Zhu, Lizhen Shi, Hai Pham, and Hao Zou. Because of their support and help, Auburn is like a home to me.

My deepest gratitude and appreciation go to my wife, my parents, my sister, and my daughter. They are the charming gardeners who help me grow strong and make my life blossom. Their love and sacrifice have paved this long journey for me to pursue my dreams.

Last but not least, I would like to acknowledge the sponsors of this research and my graduate studies at Auburn University. Particularly, I would like to acknowledge the Alabama Innovation

Award, the NASA Award NNX11AR20G, and the National Science Foundation awards 1059376, 1320016, 1340947 and 1432892.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Currently, Graphics Processing Units (GPUs) have been everywhere in our daily lives. Devices, such as desktops, laptops, tablets, smart phones, and game consoles, all have GPUs inside. Beyond the conventional functionality of graphics processing, GPUs have proven as a viable technology for a wide variety of general purpose applications to exploit the massive computing capability and high computation efficiency. In High Performance Computing (HPC), GPUs become a key performance booster to realize the realm of exascale computing. Noticeably, in the latest Top500 supercomputer list [85], 52 systems are powered by NVIDIA Tesla GPUs, such as the Titan supercomputer hosted by Oak Ridge National Laboratory, the Piz Daint supercomputer hosted by Swiss National Supercomputing Center (CSCS), Switzerland, and the Tsubame 2.5 supercomputer hosted by Tokyo Institute of Technology, Japan.

Following their success for compute-intensive high performance computing applications, GPUs have demonstrated significant performance improvements for data-intensive scientific applications such as molecular dynamics [3], document clustering [101], DNA sequence alignment [88], software router [31], and large graph processing [43]. In the meantime, the arrival of big data era has further stimulated the need of leveraging the massive computation power of GPUs in accelerating newly emerging data-intensive applications, such as data warehousing applications [5, 26, 27, 97, 34, 87] and big data processing frameworks [10, 13, 14, 79, 32]. For example, GPU appears to be an efficient vehicle for high throughput implementations of data warehousing applications; the GPU-based implementations can provide an order of magnitude or more performance improvement over traditional CPU-based implementations [34, 87]. It is reported that companies, such as Walmart, Amazon, Facebook and NASDAQ, have started to accelerate their service infrastructures using GPUs [97].

Current GPUs are often built with a Single Instruction Multiple Thread (SIMT) execution model to enable massive parallelism. Within such an execution model, threads are organized into warps and threads in a warp execute in lock-step. This execution model favors applications with few control branches and highly regular memory accesses, and generally faces two challenges, i.e., control divergence and memory divergence. Control divergence occurs when threads in a warp take different code paths for execution, while memory divergence refers to the case where intra-warp accesses cannot be coalesced into one or two cache blocks. Divergent control branches and memory accesses break the lock-step execution of threads within the same warp, severely degrading GPU resource utilization and computation throughput. For example, it is well documented that memory divergence can waste the bandwidth of long-latency memory chips. However, data-intensive applications often exhibit substantial amount of divergent memory accesses so that they are hard to be optimized for efficient GPU executions.

Even though various software-based optimization techniques can further improve the performance of GPU-accelerated data-intensive applications, they often take non-trivial efforts to revise existing code to match memory access patterns with GPU hardware characteristics. Therefore, these optimizations often include a large amount of complicated auxiliary code to manage on-chip resources and eliminate divergence in high level programming languages, such CUDA [61] and OpenCL [77]. Rogers *et al.* [70] conducted a case study of GPU programmability using two different implementations of the Sparse Matrix Multiplication from SHOC [17] benchmark suite and reported that the scalar implementation executed 2.8x less dynamic instructions than the highly optimized vector implementation. Recent works on memory divergence management for data-intensive workloads, such as CCWS [69] and DAWS [70], have demonstrated that architectural support to mitigate memory divergence is highly promising and more importantly can greatly ease the complexity in GPU programming.

This dissertation studies architectural support for memory divergence mitigation so that data-intensive applications can be efficiently executed on GPUs with reduced programming complexity.

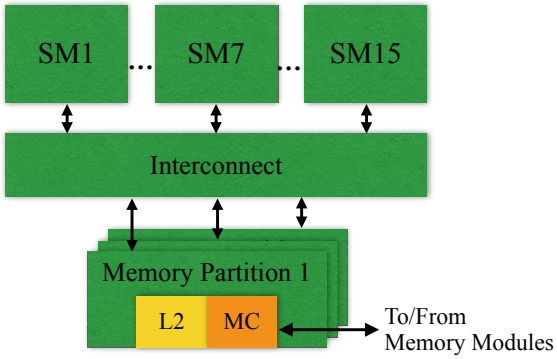


Figure 1.1: Baseline GPU Architecture.

In particular, this dissertation focuses on the impacts of divergent memory accesses on several hardware units, including cache indexing methods, cache locality management, and warp scheduling logic. The architectural deficiencies in the three units are quantitatively analyzed using motivational experiments and independently addressed with novel and light-weight techniques. This dissertation also presents comprehensive evaluations of these proposed techniques and systematic comparisons with closely related state-of-the-art techniques.

The rest of this chapter details the baseline GPU architecture studied throughout this dissertation in Section 1.1, formally defines the three architectural challenges of memory divergence in Section 1.2, summarizes the contributions of the dissertation in Section 1.3, and presents the organization of the dissection in Section 1.4.

1.1 Overview of Baseline GPU Architecture

This dissertation studies a Fermi-like baseline GPU architecture, as shown in Figure 1.1. This GPU works as a coprocessor of CPUs and is connected with CPUs via PCIe bus. Applications programmed in high level programming languages, such CUDA [61] and OpenCL [77], are first executed on CPUs. The code portions to execute in GPU cores are launched onto GPUs in the form of kernels. The data movement between CPU-side host memory and GPU device memory goes through the PCIe bus. Compared to the bandwidth of host and device memories, the bandwidth of PCIe bus is a major performance bottleneck in GPU computing.

Generally, such a discrete GPU consists of multiple Stream Multiprocessors (SMs), two uni-directional interconnection networks, several memory partition units, and a collection of off-chip memory modules. Each SM is highly multi-threaded and pipelined, i.e., each SM has a cluster of Single Instruction Multiple Thread (SIMT) cores. SIMT cores execute distinct thread, operate on scalar registers and progress in lock-step. As shown in Figure 1.2, our baseline SM mainly consists of *Operand Collector* (i.e., register file), *Execution Units (ALU/FPU/SFU)*, and *Load-Store (LD/ST)* units. LD/ST units manage accesses to various memory spaces in GPUs. According to data destination, GPU memory requests are sent to data cache (L1D), constant cache, texture cache, and shared memory respectively.

Each memory partition mainly consists of a L2 data cache portion and a memory controller (*MC*) that manages off-chip memory modules. Interconnection networks manage data movement between SMs and L2 caches, and on-chip memory channels connect L2 caches and off-chip global memory modules. The global memory is cached by the last-level L2 cache if available. GPU global memory can utilize either GDDR3/5 or DDR3 SDRAM. GDDR3/5 is similar to DDR3 in circuit organization, but GDDR3/5 can offer higher peak bandwidth than DDR3 because of its higher data transfer rate per pin and its prefetch buffers. These memory modules collectively provide high memory bandwidth to sustain the memory demand from massive parallelism, but memory operations to these memory modules have very long latency. Thus, reducing the traffic to global memory has been considered as the first principle to achieve high performance in GPUs.

We use GPGPU-Sim [4] to simulate the aforementioned baseline GPU architecture. The GPGPU-Sim 3.x Manual [1] describes other hardware units that have not been described in this dissertation in more details.

1.1.1 Warp Scheduling

As shown in Figure 1.2, each SM contains multiple physical warp slots and two warp schedulers independently manage warps with even and odd identifiers [59]. In each cycle, both warp schedulers pick one ready warp and issue its instruction into the SIMT pipeline backend [53, 59,

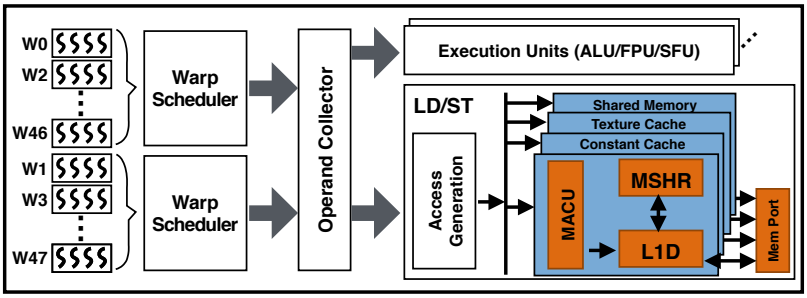


Figure 1.2: Baseline Streaming Multiprocessor (SM). Other stages of the pipeline are omitted.

60], i.e, execution units or LD/ST units. The warp scheduler is capable of zero-overhead context switches for concurrent swaps so that GPU pipeline could remain busy.

To determine the readiness of each decoded instruction, a ready bit is used to track its dependency on other instructions. It is updated in the scoreboard by comparing its source and destination registers with other in-flight instructions of the warp [8]. Instructions are ready for scheduling when their ready bits are set, i.e., data dependencies are cleared. GPU scheduling logic consists of two stages, qualification and prioritization [56]. In the qualification stage, ready warps are selected based on the ready bit that is associated with each instruction. In the prioritization stage, ready warps are prioritized for execution based on a chosen metric, such as cycle-based round-robin [56, 57, 42], warp age [69, 70], instruction age [56, 9], or other statistics that can maximize resource utilization [56]. For example, the Greedy-Then-Oldest (GTO) [69, 70] scheduler maintains the highest priority for the currently prioritized warp until it is stalled. The scheduler then selects the oldest among all ready warps for scheduling. We use GTO as the baseline warp scheduling technique in this dissertation because of its performance superiority in a larger variety of general purpose GPU benchmarks.

1.1.2 Global Memory Accesses and Memory Divergence

Once a memory instruction to global memory is issued, it is first sent to *Memory Access Coalescing Unit (MACU)* for access generation. MACU coalesces per-thread memory accesses to minimize off-chip memory traffic. For example, when 32 threads of a warp access 32 consecutive

words in a cacheline-aligned data block, MACU will only generate one memory access to L1D. Otherwise, multiple memory accesses are generated to fetch all needed data. In the rest of this dissertation, the memory instructions that incur more than 2 uncoalescable memory accesses are called divergent instructions, while the others are called coherent instructions.

The resultant memory accesses from MACU are sequentially sent to L1D via a single 128-byte port [9]. For a load access, if it hits in L1D, the requested data is written back to the register file; if it misses in L1D, one demand request is generated to fetch data from lower memory hierarchy. The Missing Status Holding Register (MSHR) is used to track in-flight memory requests and merge duplicate requests to the same cache line. After MSHR allocation, a memory request is buffered into the *Memory Port* for network transfer. An MSHR entry is deallocated after its corresponding memory request is back and all accesses to that block are serviced. Memory requests buffered in the memory port are drained by the on-chip network in each cycle when lower memory hierarchy is not saturated.

L1D does not support coherence, so it evicts cache blocks on stores to global memory. Stores require no MSHR and are directly buffered into the memory port.

1.2 Challenges of Memory Divergence

It has been well documented that divergent memory accesses can waste the bandwidth of on-chip network and off-chip memory channels. This dissertation studies three new architectural challenges that are associated with memory divergence.

1.2.1 Intra-Warp Associativity Contention

Since GPUs normally have a very small L1D, any potential locality could be easily thrashed by the aggregated memory demands from the massive parallelism, especially when memory divergence boosts the per-warp cache footprint. However, memory divergence incurs not only inter-warp capacity misses, but also high intra-warp associativity conflict misses when the divergent

intra-warp accesses are pathologically concentrated into a few cache sets. Because of the discrepancy between low cache associativity and high concentration of divergent intra-warp accesses, a warp can have consecutive intra-warp associativity conflicts, resulting in repetitive execution stalls. Such intra-warp conflicts can cause execution stalls to more warps, hindering possible overlaps of memory and computation for high instruction throughput.

1.2.2 Partial Caching

Within the lock-step execution model, a warp becomes ready when all of its demanded data is available; warps that have missing data, regardless of the data size, are excluded for execution. This execution model of GPU expects that all cache lines of each divergent load instruction are cached as a unit when there is locality. However, conventional cache management is unaware of the GPU execution model and the collective nature of divergent memory blocks. As a result, some blocks of a divergent load can be evicted while others are still cached, resulting in a varying number of cache misses for individual loads. Coherent loads can also experience this kind of varying number of cache misses when they issue two memory accesses each time. Thus, **partial caching** refers to the scenario where a load instruction has part of its data items hit in the cache and others missed from a single issuance.

1.2.3 Memory Occlusion

MSHR is often implemented as a fully-associative structure and thus is limited by capacity. This leads to a structural hazard due to the mismatch between limited memory-level parallelism (MLP) and massive thread-level parallelism (TLP). This hazard can be exaggerated by bursty cache accesses from divergent loads. As shown in the right part of Figure 1.3, the coherent load that has one cache miss can be immediately serviced, while the four uncoalescable memory accesses of the divergent load suffer from insufficient MSHR entries because only two MSHR entries are available. In this example, the access to block *F* that misses in L1D is replayed until memory request of block *A* is back and its MSHR entry is deallocated. During the access replay, the currently prioritized

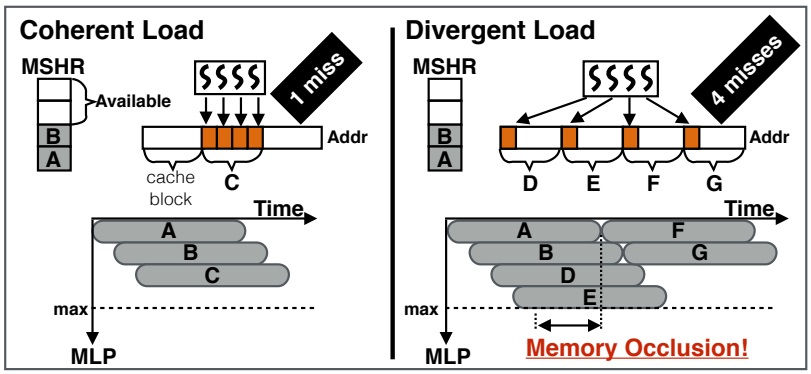


Figure 1.3: Illustrative example of memory occlusion. The MSHR of L1D has 4 entries, two of which have been allocated to track the outstanding memory requests of cache blocks A and B. The remaining MSHR entries are sufficient for a coherent load, but occlude the divergent load so that demand requests of blocks F and G are replayed.

memory instruction can not make progress, occluding L1D and preventing other ready memory instructions from accessing LD/ST units. We refer to such a scenario as ***Memory Occlusion***. Memory Occlusion degrades memory instruction throughput in LD/ST units and prevents other memory instructions that do not need MSHR from accessing L1D.

1.3 Research Contributions

In this dissertation, we have thoroughly investigated three architectural challenges that can severely impact the performance of using GPUs to accelerate memory divergent benchmarks. The three challenges lie in cache indexing method, cache locality management, and warp scheduling logic. For each challenge, this dissertation proposes a solution and compares it with the closely related state-of-the-art techniques. In particular, this dissertation has made following three contributions.

1.3.1 Eliminating Intra-Warp Conflict Misses in GPU

Cache indexing functions play a key role in reducing conflict misses by spreading accesses evenly among all sets of cache blocks. Although various methods have been proposed, no significant effort has been expended on the behavior of conflict misses in GPU where threads are

organized into warps and execute in lock-step. When memory divergence happens, a warp incurs up to warp-size (e.g., 32) independent cache accesses. Such a burst of divergent accesses not only increases contention on cache capacity, but also incurs intra-warp associativity conflicts when they are pathologically concentrated in a few cache sets. Due to the lock-step execution, the LD/ST units would be stalled when intra-warp concentration exceeds available cache associativity. Through an in-depth analysis of GPU access patterns, we find that column-majored strided accesses are likely to incur high intra-warp concentration. Based on the analysis, we propose a *Full Permutation (FUP)* based indexing method that adapts to both large and medium strides in this pattern. Across the 10 highly cache-sensitive GPU applications we have evaluated, FUP eliminates intra-warp associativity conflicts and outperforms two state-of-the-art indexing methods, bitwise-XOR [25] and prime displacement [47], by 22% and 15%, respectively.

1.3.2 DaCache: Memory Divergence-Aware GPU Cache Management

The lock-step execution model of GPU requires a warp to have the data blocks for all its threads before execution. However, there is a lack of salient cache mechanisms that can recognize the need of managing GPU cache blocks at the warp level for increasing the number of warps ready for execution. In addition, warp scheduling is very important for GPU-specific cache management to reduce both intra- and inter-warp conflicts and maximize data locality. To solve this challenge, we propose a Divergence-Aware Cache (*DaCache*) management that can orchestrate L1D cache management and warp scheduling together for GPGPUs. In DaCache, the insertion position of an incoming data block depends on the fetching warp's scheduling priority. Blocks of warps with lower priorities are inserted closer to the LRU position of the LRU-chain so that they have shorter lifetime in cache. This fine-grained insertion policy is extended to prioritize coherent loads over divergent loads so that coherent loads are less vulnerable to both inter- and intra-warp thrashing. DaCache also adopts a constrained replacement policy with L1D bypassing to sustain a good supply of Fully Cached Warps (FCW), along with a dynamic mechanism to adjust FCW during runtime. Our experiments demonstrate that DaCache achieves 40.4% performance improvement over

the baseline GPU and outperforms two state-of-the-art thrashing-resistant techniques, RRIP [38] and DIP [64], by 40% and 24.9%, respectively.

1.3.3 OAWS: Memory Occlusion Aware Warp Scheduling

GPUs deliver massive computation parallelism by alternating the execution of many warps and overlapping the memory accesses of some warps with the computations of other warps. However, the execution of concurrent warps are often disrupted by various hazardous situations, such as control and memory divergences, which present a significant impediment to GPU performance and have attracted a lot of research interest. To solve this challenge, we have closely examined GPU resource utilization when executing memory-intensive benchmarks. Our detailed analysis of GPU global memory accesses reveals that divergent load instructions can easily incur memory occlusion. Such memory occlusion prevents other ready memory instructions from accessing L1 data cache, eventually stalling warp schedulers and degrading the overall performance. We have designed Memory Occlusion Aware Warp Scheduling (OAWS) that can dynamically predict the demand of MSHR entries of divergent memory instructions and maintain such a concurrency that the aggregate MSHR consumption from all active warps is within the MSHR capacity, thereby preventing memory occlusions. Our experimental results show that the static and dynamic versions of OAWS achieve 35.3% and 74% performance improvement, compared to baseline GTO warp scheduling. The dynamic OAWS outperforms MASCAR [73], CCWS [69], and SWL-Best [69] by 65.8%, 57.2%, and 8.5%, respectively.

1.4 Dissertation Overview

In the rest of this dissertation, we detail three architectural challenges that are associated with divergent memory access patterns in GPUs, and then provide detailed descriptions for our techniques. Each chapter focuses on one challenge, along with comprehensive performance evaluations and comparisons with contemporary state-of-the-art techniques.

In Chapter 2, we systematically examine the three challenges that prevent memory divergent benchmarks from gaining high performance on GPUs to motivate our innovations.

In Chapter 3, we introduce a Full-Permutation based GPU cache indexing to eliminate intra-warp conflict misses due to pathological behaviors in existing cache indexing methods. Our performance evaluation demonstrates that our technique can distribute divergent intra-warp accesses into all available cache sets and improve the performance of memory divergent benchmarks.

In Chapter 4, we introduce a memory divergence-aware GPU cache management technique that leverages both warp scheduling prioritization and memory divergence characteristics to make caching decisions at instruction level. Our experiments demonstrate that orchestrating warp scheduling and cache management can better preserve intra-warp locality and resist both inter- and intra-warp thrashing.

In Chapter 5, we introduce a light-weight scheduling technique to source-throttle memory occlusion, a structural hazard caused by the mismatch between limited capacity of Missing Status Holding Registers (MSHRs) and bursty requests from divergent memory accesses. We propose both static and dynamic methods to predict the MSHR consumption of divergent load instructions and a new qualification metric for scheduling logic to proactively prevent memory occlusion from occurring. Our experiments demonstrate that both static and dynamic prediction methods can effectively reduce stall cycles in LD/ST units as well as in warp schedulers and outperform three state-of-the-art warp scheduling techniques.

Eventually, we conclude this dissertation and outline two promising opportunities as future work in Chapter 6.

Chapter 2

Problem Statement

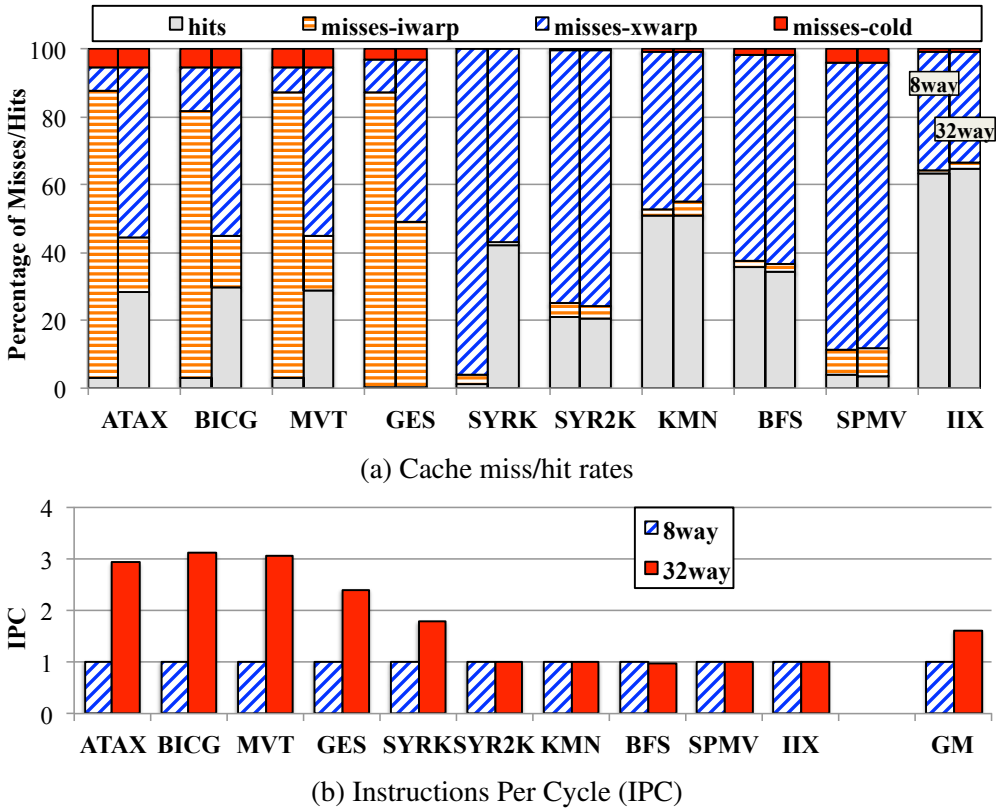
This chapter presents motivational experiments that are designed to reveal the problems of intra-warp associativity conflicts, partial caching, and memory occlusion and quantify their direct impacts on performance and on-chip resource utilization. In Section 2.1, the impacts of associativity conflicts on performance are demonstrated, and a common memory access pattern that is prone to cause associativity conflicts in GPUs is analyzed. In Section 2.2, partial caching is quantified using a new metric, Misses Per Load Instruction, and the interplay between cache contention and warp scheduling is analyzed to motive our solution. In Section 2.3, the impacts of memory occlusion are quantified in terms of stalled cycles in LD/ST units and warp schedulers and the occlusion time in L1D access latency, and the predictability of MSHR consumption under the baseline Greedy-Then-Oldest (GTO) warp scheduling is examined using a representative memory divergent benchmark.

2.1 Intra-Warp Conflict Misses due to Pathological Cache Indexing Method

In order to illustrate the problem of intra-warp associativity conflicts and quantify their direct impacts on GPU performance, we use two L1D configurations that have the same total capacity (32KB) but different cache associativities (8 v.s. 32) to execute 10 highly cache sensitive benchmarks. The details of the benchmarks and the configuration parameters of GPGPU-Sim that are used in this experiment are presented in Section 3.3 and Table 3.1, respectively.

2.1.1 Types of Cache Misses in GPU

For simplicity, we categorize cache misses into cold (*misses-cold*), intra-warp (*misses-iwarp*) and inter-warp (*misses-xwarp*) misses [40]. Meanwhile, we also present the percentages of cache



(a) Cache miss/hit rates

(b) Instructions Per Cycle (IPC)

Figure 2.1: The impacts of cache associativity on highly cache-sensitive benchmarks. Both caches have 32KB capacity and 128B lines.

hits (*HIT*). An intra-warp miss refers to the case where a thread's data is evicted by other threads within the same warp, otherwise a conflict miss is referred to as inter-warp miss. In this experiment, intra-warp misses are typically correlated with associativity conflicts. As shown in Figure 2.1a, after increasing the associativity from 8 to 32, the intra-warp misses (*misses-iwarp*) in ATAX, BICG, MVT, and GES are significantly reduced. 49% of the misses in GES are still intra-warp misses, because it has two fully divergent loads that contend for L1D capacity. Meanwhile, larger associativity reduces the inter-warp misses (*misses-xwarp*) in SYRK, indicating that associativity conflicts can also occur between inter-warp accesses. Figure 2.1b presents the impacts of cache associativity on the IPC of the 10 benchmarks. For ATAX, BICG, MVT, GES, and SYRK, a 32-way 32KB L1D improves performance by 2.6×. Even though a 32-way cache is impractical for real GPU architectures, this experiment shows that eliminating associativity conflicts are critical to sustain high performance for memory divergent benchmarks.

2.1.2 Associativity-Sensitive Access Patterns

With multidimensional data arrays, the **column-major strided** access pattern is prone to create high intra-warp contention on associativity. The most common example of this pattern is A[tid*STRIDE+offset], where *tid* is the unique thread ID and *STRIDE* is user-defined stride size. By using this pattern, each thread iterates a stride of data independently. In a conventional cache indexing function, the target set is computed as $set = (addr/blkSz)\ mod\ n_{set}$, where $addr$ is the memory address, $blkSz$ is the length of cache block and n_{set} is the number of cache sets. When the address stride between two consecutive threads is equal to a multiple of $blkSz \times n_{set}$, all blocks needed by a single warp are mapped into the same cache set. For example, when the stride size (*STRIDE*) is 4096 bytes, the two consecutive intra-warp memory addresses, 0x80000000 and 0x80001000, will be mapped into the set 0 in our baseline L1D that has 32 cache sets and 128B cache lines.

Since cache associativity is often smaller than warp size (32), associativity conflicts occur within each single divergent load and then the memory pipeline is congested by the burst of intra-warp accesses. One existing work [40] uses aggressive L1D bypassing to alleviate associativity conflicts, but leaves L1D capacity under-utilized. By contrast, we investigate the cache indexing method to dispense the bursty intra-warp access into all cache sets so that intra-warp contention is reduced and cache capacity can be better utilized.

2.2 Partial Caching of Divergent Load Instructions in GPU

In this section, we use 20 data intensive benchmarks from Rodinia [12], SHOC [17], Poly-Bench/GPU [28], and MapReduce [32] to reveal and analyze the problem of partial caching in baseline LRU caches. For each benchmark, Table 4.2 lists a brief description and the input size that we use in following motivational experiments. The benchmarks are categorized into memory-divergent and memory-coherent ones, depending on the dynamic memory divergence of load instructions in these benchmarks. In general, memory-divergent benchmarks are more sensitive to cache capacity than memory-coherent benchmarks. Recent work [69, 70, 40, 90] reports that high

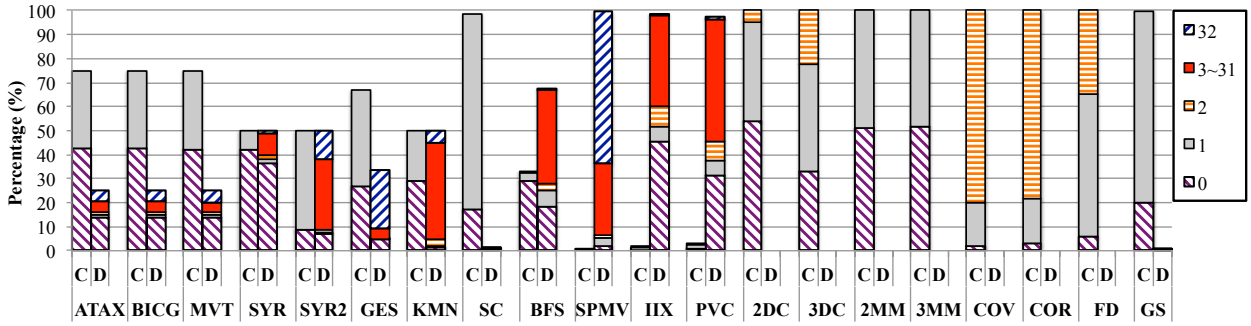


Figure 2.2: Distribution of Misses Per Load Instruction (MPLI) in L1 data cache. MPLIs are categorized into five groups: 0 (MPLI=0), 1 (MPLI=1), 2 (MPLI=2), 3∼31 (3⩽ MPLI ⩽ 31), and 32 (MPLI=32). MPLIs for coherent (*C*) and divergent (*D*) load instructions are accumulated separately. Each of the benchmarks on the right of the figure has only *C* bar for coherent instructions.

intra-warp L1D locality exists among these cache-sensitive workloads. In addition, BFS, SPMV, IIX, and PVC also have rich branch divergence. The GPGPU-Sim configuration parameters used in following experiments are presented Table 4.1. In order to isolate the impacts of intra-warp associativity conflicts from the investigation of partial caching, we use the indexing method from real Fermi GPUs, pseudo-random hashing function [58], to wide spread intra-warp memory accesses. This change in GPGPU-Sim configuration gives a faithful simulation of real GPU hardware.

2.2.1 Partial Caching of Divergent Memory Accesses

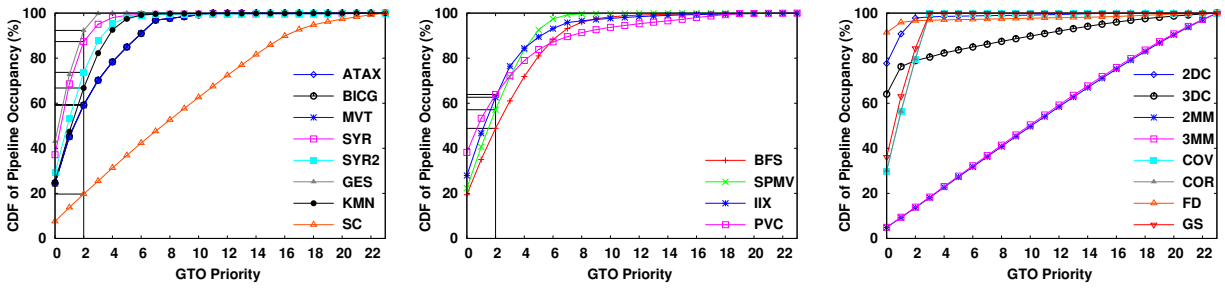
Metrics, such as Miss Rate and Misses Per Kilo Instructions (MPKI), are often used to evaluate the performance of cache management. In view of the wide variation of cache misses per instruction, we use **Misses Per Load Instruction (MPLI)** to quantify partial caching in GPU L1D. Divergent load instructions that have MPLIs in the range from 1 to $\{Req(pc,w)-1\}$ are considered as being partially cached, where $Req(pc,w)$ is the number of cache accesses that warp w incurs at memory instruction pc. For divergent load instructions in our baseline GPU, MPLI is typically in the range from 0 to 32. MPLI of 0 indicates that a load instruction has no cache misses and is considered as being fully cached; MPLI of 32 indicates that a load instruction has no cache hits and is considered as being fully missed; any MPLI value between 1 and 31 indicates a partially cached instruction. Meanwhile, MPLI is in the range from 0 to 2 for coherent load instructions.

MPLI can be calculated by counting the number of cache misses a load instruction experiences after all of its memory accesses are serviced by L1D.

Figure 2.2 shows the distribution of MPLIs across the 20 GPGPU benchmarks we have evaluated in this study. For simplicity, MPLIs are categorized into five groups. For divergent loads, the two categories of *2* (MPLI=2) and *3∼31* ($3 \leqslant$ MPLI $\leqslant 31$) in the figure, together describe the existence of partial caching. Note that this range can only provide a close approximation for partial caching because branch divergence can reduce the number of uncoalescable memory accesses a divergent load can generate. For example, a warp with 16 active threads can maximally generate 16 memory accesses for a divergent load, and an MPLI of 16 indicates full caching for this load of the warp, while an MPLI of 16 often indicates partial caching for warps with 32 threads. As we can see from the figure, coherent loads of the memory-divergent benchmarks do not experience the problem of partial caching because they all generate one memory access per instruction. However, divergent load instructions in these benchmarks greatly suffer from partial caching. Substantial amount of divergent loads in SYR2, KMN, BFS, SPMV, IIX, and PVC are partially cached. Memory-coherent benchmarks, such as 2DC, 3DC, COV, COR, and FD, also experience partial caching (MPLI=1), because their load instructions generate two memory accesses each time. Such prevalent partial caching comes from massive parallelism and divergence-oblivious cache management, exacerbates inter-warp contention on limited L1D capacity, and results in early evictions of cache lines after being used only once.

2.2.2 Warp scheduling and Cache Contention

In view of the severe cache misses as discussed in Section 2.2.1, we have further examined the impact of warp scheduling on L1D contention. GPU warp scheduling is often driven by a prioritization scheme as introduced in Section 1.1.2. For example, in the baseline Greedy-Then-Oldest (GTO) warp scheduling, warps are dynamically prioritized by their "ages", and older warps are preferentially prioritized at runtime. In order to quantify the cache contention due to aggressive warp scheduling, we measure the occupancy of warp schedulers by all active warps. Figure 2.3



(a) Memory-Divergent (b) Memory- and Branch-Divergent (c) Memory-Coherent

Figure 2.3: The CDF of warp scheduler occupancy by all active warps. The percentage reflects the frequency that each warp is scheduled. *GTO priority* refers to the "age" of each warp. Since each of the two warp schedulers in an SM manages 24 warps, 0 represents the highest priority, while 23 represents the lowest priority. Our baseline L1D can typically accommodate three divergent warps for each warp scheduler.

shows the Cumulative Distribution Function (CDF) of warp scheduler occupancy when the evaluated benchmarks are scheduled under GTO prioritization. Typically, these benchmarks have one fully divergent load (resulting in 32 accesses) and one coherent load (resulting in one access) in the kernel, so the cache footprint of each warp is 33 cache lines at runtime. Since the two warp schedulers in each SM of the baseline GPU have equal access to the shared L1D (32KB, 256 cache lines), enabling four warps for each warp scheduler will over-subscribe L1D capacity (264 cache lines), causing inter-warp contention. Thus, only three warps can be fully cached for each warp scheduler, leading to an aggregated utilization of 77% of the L1D capacity. With 0 representing the highest GTO priority, fully caching such a small amount of warps means that L1D will inevitably be thrashed if the warps with GTO priorities lower than 3 are scheduled.

For memory-divergent benchmarks in Figure 2.3a, 58%∼91% of the total cycles are occupied by the top 3 prioritized warps. Even though warps with lower priorities are infrequently scheduled, they are highly likely to thrash the locality of other warps with higher priorities whenever they are scheduled. Meanwhile, the cache lines of these low-priority warps are often evicted before any re-reference, even though intra-warp locality is high. Since branch divergence reduces the number of accesses a divergent load can generate, more warps can be fully cached for benchmarks with both memory- and branch-divergence. As shown in Figure 2.3b, the occupancy drops to 48%∼63% among these benchmarks. Such variation in warp scheduling incurs immediate cache conflicts.

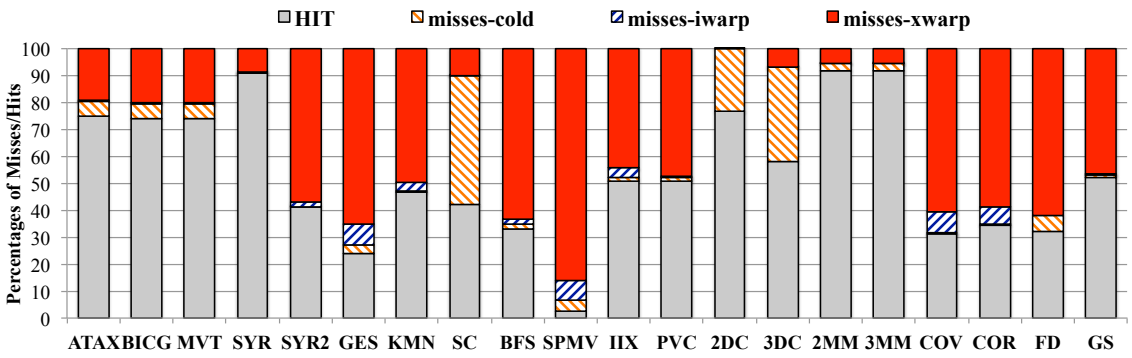


Figure 2.4: Categorization of L1D thrashing.

As in the categorization method described in Section 2.1.1, we categorize caches misses into cold (*misses-cold*), intra-warp (*misses-iwarp*) and inter-warp (*misses-xwarp*) misses. Figure 2.4 shows that the majority of cache misses are due to inter-warp conflicts, which in turn cause high MPLI as shown in Figure 2.2 and varied occupancy of warp schedulers as shown in Figure 2.3.

2.3 Memory Occlusion in Data-Intensive GPGPU Workloads

This section investigates the impacts of memory occlusion on GPU performance. First, we breakdown LD/ST stalls cycles to reveal the impact of memory occlusion on the utilization of LD/ST units and warp schedulers. Second, we quantify how memory occlusion impacts global memory access time. Finally, we use a case study to demonstrate that MSHR consumption, the cause of memory occlusion, is predictable under GTO warp scheduling. The configuration parameters for GPGPU-Sim and the details of the benchmarks that we use in following experiments are presented in Table 5.1 and Table 5.2, respectively.

2.3.1 Stalls in LD/ST Units and Warp Schedulers

When LD/ST units are stalled, a ready memory instruction can not be issued. We refer to such stall cycles as LD/ST stall cycles. Besides MSHR unavailability and memory port congestion, sequentially processing uncoalescable memory accesses makes the LD/ST units unavailable to warp schedulers and delays other ready memory instructions accessing L1D, even if current

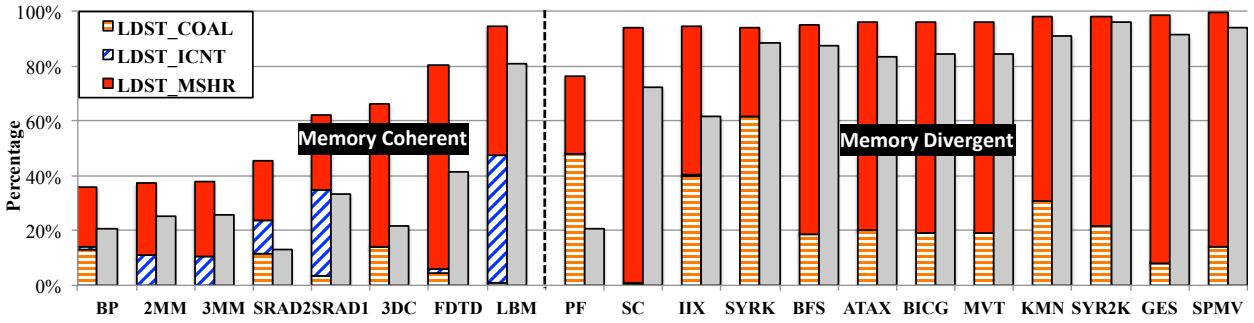


Figure 2.5: The breakdown of LD/ST stall cycles (the stacked bar on left) and the percentage of cycles for which the warp schedulers are stalled due to LD/ST stalls (the right bar). The dotted line divides all benchmarks into memory coherent (left) and memory divergent (right) ones. Benchmark characteristics and simulator details are summarized in Section 5.5.1.

memory instruction is making progress to send out its memory accesses. According to the three causes that can stall LD/ST units, we breakdown the LD/ST stall cycles into three categories in Figure 2.5: 1) coalescing stalls (*LDST_COAL*) — when L1D has successfully serviced one uncoalescable memory access, no matter if it is a cache hit or miss; 2) MSHR stalls (*LDST_MSHR*) — when a cache miss can not be processed due to MSHR unavailability; and 3) ICNT stalls (*LDST_ICNT*) — when a cache miss can not be processed due to on-chip network congestion, but MSHR entries are available. Among the memory coherent benchmarks, 2MM, 3MM, SRAD1, SRAD2, and LBM experience a large percentage of *LDST_ICNT* stalls. The five benchmarks write significant amounts of data into global memory, congesting the network from SM to L2 cache. For memory divergent benchmarks, *LDST_MSHR* dominates LD/ST stall cycles, with an average of 66% of total cycles waiting for MSHR entries. These divergent benchmarks are read-intensive, and read requests impose limited pressure on the network from SM to L2 cache, so *LDST_ICNT* plays a negligible role in these benchmarks.

Although LD/ST units are stalled, warp schedulers can still issue computation instructions into execution units to overlap the stalls in LD/ST units. The capability of overlapping LD/ST stalls explains why warp schedulers (the gray bar) are stalled less than LD/ST units across all of the benchmarks in Figure 2.5. However, LD/ST stalls eventually idle warp schedulers when all warps are waiting to be scheduled to issue memory instructions. For memory coherent benchmarks, on

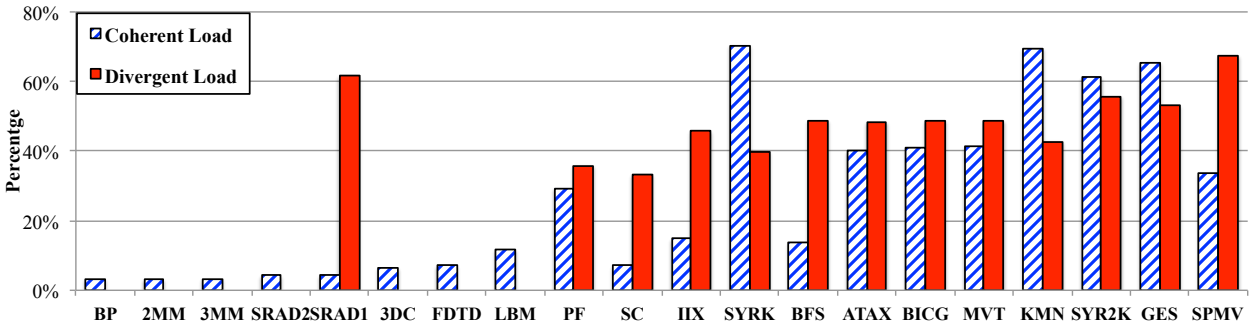


Figure 2.6: The percentage of memory occlusion time in the memory access latency for coherent and divergent loads.

average, warp schedulers waste 28% of the total cycles waiting for the availability of LD/ST units. This percentage increases to 75% for memory divergent benchmarks. Such high stalls in warp schedulers directly lead to severe degradation of instruction throughput.

2.3.2 Quantifying Memory Occlusion Time

Without memory occlusion, an issued memory instruction with N accesses should retire from LD/ST units after N cycles plus the latency of register file and LD/ST units. Here, N cycles are needed by L1D to sequentially process all of the N accesses. Any extra cycle is counted as the delay caused by memory occlusion. We define these extra cycles as instruction delay. Consequently, L1D access latency can be divided into memory occlusion time and L1D hit/miss time. In order to quantify the impacts of memory occlusion, we compare such delays with the memory access time of load instructions. In this study, the L1D access time of each load instruction starts when it is issued by warp scheduler and ends when all of its needed data is written back to register file.

Figure 2.6 shows the percentage of memory occlusion time in the average L1D access latency for both coherent and divergent load instructions across all the data intensive benchmarks. On average, memory occlusion delays account for 4% of L1D access time for all memory coherent benchmarks. SRAD1 has very few divergent loads through its 502 kernel invocations. In memory divergent benchmarks, memory occlusion delays reach 33% and 47% for coherent and divergent loads, respectively. These large delays dramatically prolong memory access time, demanding a

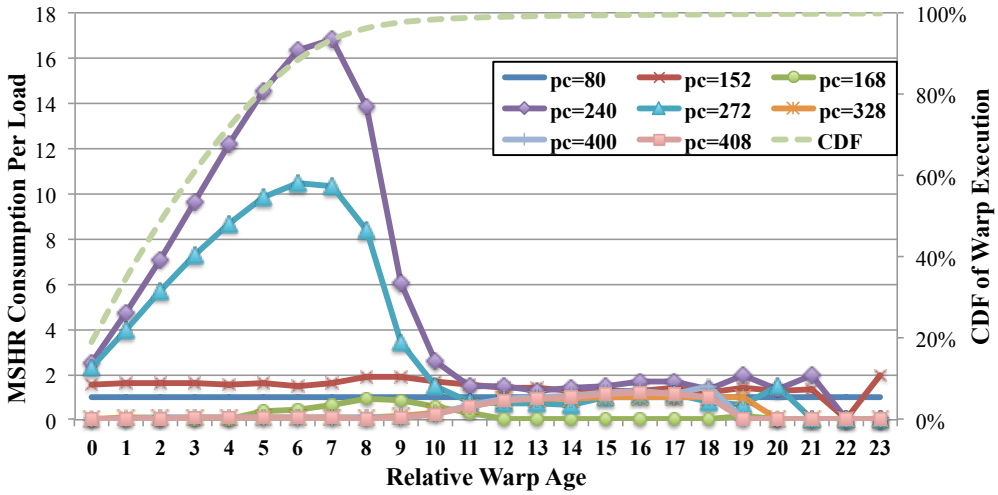


Figure 2.7: The MSHR consumption of each load instruction in BFS kernel and the CDF of warp execution times when using GTO scheduler. *pc=240* and *pc=272* are the two divergent loads. Age 0 represents the oldest warp.

higher degree of computation-memory overlap. However, these memory-divergent benchmarks often lack sufficient computation instructions, causing the warp scheduler stalls shown in Figure 2.5. It is clear that memory occlusion is a performance destructor for memory divergent GPGPU workloads, which is the third challenge we study in this dissertation.

2.3.3 Predictability of MSHR consumption

Since the depletion of MSHR entries is the cause of memory occlusion, we use one kernel from BFS to study the predictability of MSHR consumption when using a GTO warp scheduler. This kernel has 2 divergent load instructions (pc=240 and pc=272), and 6 other coherent loads. As shown in Figure 2.7, the MSHR consumption levels of the two divergent loads are linear to each warp's relative age. Since the oldest warp (age=0) is constantly prioritized under GTO scheduling, it has the lowest MSHR consumption, while the warp of age 7 has the highest MSHR consumption. The younger warps (age>7) have lower MSHR consumption because they are only scheduled when older warps are retiring. The CDF of warp execution times shows that the oldest 8 warps (age=0-7) occupy 94% of total warp scheduler cycles. This observation also applies to the other

memory divergent benchmarks we have evaluated. Thus, we can conclude that a load instruction's MSHR consumption is highly correlated with its owner warp's scheduling priority.

2.4 Summary

In summary, this chapter has analyzed the direct impacts of intra-warp associativity conflicts, partial caching, and memory occlusion. Therefore, this dissertation is devoted to addressing these issues, preparing future GPU architectures to efficiently execute memory-divergent benchmarks. To be more specific, this dissertation seeks architectural optimization to tackle above issues from the following three directions:

- Using concentration-resistant cache indexing method to uniformly distribute GPU memory accesses into all available cache sets
- Leveraging warp scheduling logic and memory divergence characteristics to reduce partial caching and resist inter- and intra-warp cache thrashing
- Predicting the consumption of individual divergent load instructions and then scheduling instructions that will not incur memory occlusion

The solutions this dissertation presents are:

- Full-permutation Based GPU Cache Indexing (in Chapter 3)
- DaCache: Memory Divergence-Aware GPU Cache Management (in Chapter 4)
- OAWS: Memory Occlusion Aware Warp Scheduling (in Chapter 5)

Chapter 3

Eliminating Intra-Warp Conflict Misses in GPU

3.1 Introduction

Recently, GPUs have employed a hierarchy of date caches, which can reduce the latency of memory operations and save the on-chip network and off-chip memory bandwidth when there is locality within the accesses. However, the contention from massive parallelism often makes the caching performance unpredictable. Notably, the bursty divergent accesses can cause associativity stalls when they are pathologically concentrated into a few cache sets as described in Section 1.2.1 and Section 2.1. To tackle this problem, MRPB [40] aggressively bypasses L1D whenever associativity stall occurs, but the cache capacity is still underutilized. Two recent works [69, 70] have reported that throttling the number of actively scheduled warps is able to reduce the accumulated working set so that the contention on cache capacity is alleviated and locality is preserved. However, they are mainly designed to alleviate capacity misses, having little control over intra-warp associativity conflicts. Without spreading bursty intra-warp accesses evenly into all cache sets, associativity conflicts inevitably undercut the potential performance benefits that other optimizations can bring. For example, concurrency throttling techniques, such as CCWS [69] and DAWS [70], become futile when intra-warp associativity conflicts are high.

Pseudo-random cache indexing methods have been extensively studied to reduce conflict misses within CPU systems. However, no prior indexing method has exploited the pathological behaviors of GPU cache indexing. In CPU systems, parallelism is supported at a moderate level, in which memory accesses are often dispersed over time. However for GPUs, high thread counts are common, intra-warp accesses often come in long bursts when memory divergence occurs and then memory bandwidth utilization plays a critical role in sustaining high computation throughput. These distinctive features pose challenges in designing a GPU-specific indexing method.

Based on these observations and the motivational results in Section 2.1, we study how to design a GPU data cache indexing method to eliminate intra-warp associativity conflicts. Our contributions from this study include:

- Presenting the problem of intra-warp conflict misses in GPU from the aspect of pathological behaviors of current cache indexing method;
- Proposing a new metric, *intra-warp concentration*, to evaluate GPU cache indexing methods. This metric quantifies the dynamic concentration of divergent intra-warp accesses into cache sets and is more correlated with intra-warp conflicts;
- and designing a *Full-Permutation (FUP)* based GPU cache indexing method, which achieves perfect intra-warp concentration for strided access patterns among GPU benchmarks and significantly reduces the conflict misses due to intra-warp contention.

Our experimental results show that FUP improves the performance of 10 highly cache-sensitive GPU benchmarks by 2.46× (Geometric Mean), and outperforms two state-of-the-art cache indexing methods, bitwise-XOR [25] and prime displacement [47], by 22% and 15%, respectively. The metric of intra-warp concentration is also proven to be more closely correlated with the quality of GPU cache indexing methods than other GPU-oblivious static metrics.

The rest of chapter is organized as follows: Section 3.2 details the design of FUP, experimental results and related work are presented in Section 3.3 and Section 3.4, respectively, and Section 3.5 summarizes this chapter.

3.2 Full-permutation Based GPU Cache Indexing

In this section, we first elaborate a new metric for quantifying the distribution of intra-warp accesses and then propose our full-permutation indexing method.

3.2.1 A Metric for GPU Cache Indexing Method

We use **Intra-warp Concentration** to quantify the distribution uniformity of intra-warp accesses into the cache sets. It is measured by:

$$Intra\text{-}warp\ concentration = \frac{N_{acc}}{N_{cache\_sets\_touched}} \tag{3.1}$$

where N_{acc} is the number of accesses in a load instruction and $N_{cache\_sets\_touched}$ is the number of sets that are caching the data of the load instruction. Within our baseline GPU, a warp can maximally generate 32 divergent accesses and the L1D has 32 cache set; an intra-warp concentration of 1 indicates an ideal distribution of intra-warp accesses, i.e., intra-warp contention on cache associativity is eliminated. Any value larger than 1 indicates the existence of intra-warp concentration. An intra-warp concentration of 32 is the worst case where all the intra-warp accesses are concentrated into the same cache set. In Section 3.3, we will see several of the benchmarks we have evaluated constantly experience the highest concentration under our baseline GPU cache indexing method. Note that coherent loads often lead to an ideal intra-warp concentration.

Different from the static metrics used in [47] to quantify the pathological behaviors of cache indexing methods of CPU systems, intra-warp concentration describes dynamic contention among bursty intra-warp accesses. Since avoiding long memory accesses caused by intra-warp contention is very critical, it is beneficial to introduce the intra-warp concentration metric for measuring the effectiveness of conflict management by GPU cache indexing methods.

3.2.2 Feature Bits of Intra- and Inter-Warp Addresses

In order to disperse intra-warp accesses into all cache sets, it is necessary to understand how one address is different from the others. In the column-major strided access pattern as described in Section 2.1.2, the majority of the bits in intra-warp addresses are the same, i.e., having no variability, while a small amount of bits are altered. We name those altered positions as *Feature Bits*. The length of feature bits depends on the warp size. For example, when warp size is 32, a

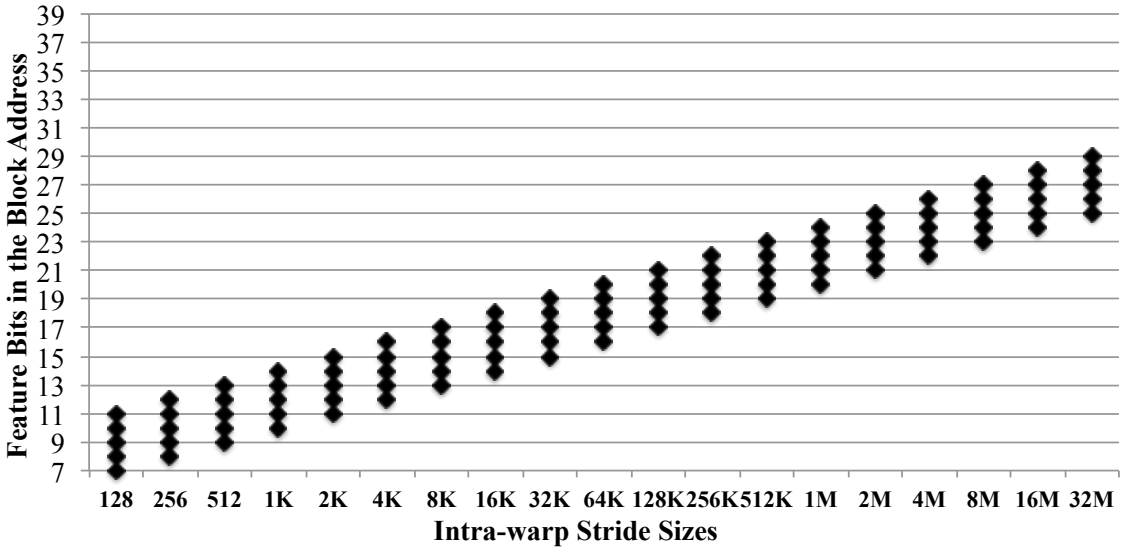


Figure 3.1: The feature bits in the block addresses when various strides are used in the strided access of *tid*STRIDE*. Here $tid \in [0, 31]$ and the bits of block offsets ($0 \sim 6^{th}$) are omitted.

warp would generate up to 32 accesses; consequently, $log_2(Warp\text{-}size)$ bits in the block addresses become feature bits to distinguish intra-warp accesses.

In order to illustrate the distributions of feature bits, we conduct a case study using the warp in which threads have global indices between 0 and 31. We denote ths warp as $Warp_0$. Figure 3.1 shows the positions of feature bits in the intra-warp block addresses of $Warp_0$ when the stride sizes range from 128 to 32M. Assuming a 40-bit virtual address space, the feature bits of $Warp_0$'s accesses spread in the range from 7^{th} to 29^{th} bit. Since a GPU kernel is often launched with a numerous number of threads, this upper bound would be theoretically extended close to the most significant bit of the block address. Given that the global memory size is no larger than 16GB among contemporary GPGPU cards, the range from 7^{th} to 34^{th} bit covers any legitimate stride size. Thus, we use bits in this range ($num\_feature\_bits$) to search feature bits for GPU cache indexing. We will show how feature bits can be used in a GPU cache indexing method to spread intra-warp accesses evenly into cache sets. Compared to the designated feature bits, if the invariable bits are used to form the set index, identical bits exist in the resultant set indexes of intra-warp accesses, causing intra-warp concentration.

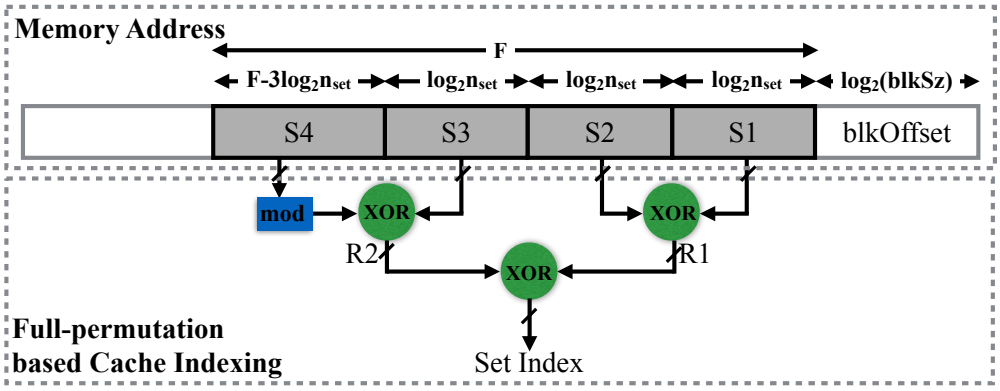


Figure 3.2: Illustration of full-permutation cache indexing.

3.2.3 Full-Permutation for GPU Cache Indexing

In order to cover all feature bits in memory addresses, we propose the *Full-Permutation (FUP)* based cache indexing method. In general, FUP uses F bits in the middle of block addresses, where $F = max(num\_feature\_bits, 4 \times log_2 n_{set})$. As shown in Figure 3.2, the F bits are divided into four groups, i.e., *S1, S2, S3,* and *S4*. We now discuss how L1D impacts the implementation of FUP:

1. When $num\_feature\_bits$ is equal to $4 \times log_2 n_{set}$, each group has $log_2 n_{set}$. Thus, the four groups of bits are paired, and then XORed in parallel at the first level, and then the two intermediate indexes, *R1* and *R2*, are further XORed to generate the final index.

2. When $num\_feature\_bits$ is smaller than $4 \times log_2 n_{set}$, FUP could still be implemented as case 1. Over-subscribing the bits of block addresses for cache indexing simplifies the hardware implementation and can disperse inter-warp accesses into all sets.

3. When $num\_feature\_bits$ is larger than $4 \times log_2 n_{set}$, *S4* has more bits than the other three groups. We use a prime number based modulo operation (*mod*) to convert *S4* into $log_2 n_{set}$ bits so that the remaining logic could be unchanged. This modulo operation could be implemented using a set of narrow add operations [47] and causes limited intra-warp concentration for very large strides.

In total, this scheme requires $3 \times log_2 n_{set}$ two-input XOR gates for the logic implementation. The delay of two-level XOR gates is less than 1 cycle even for a very aggressively pipelined processor.

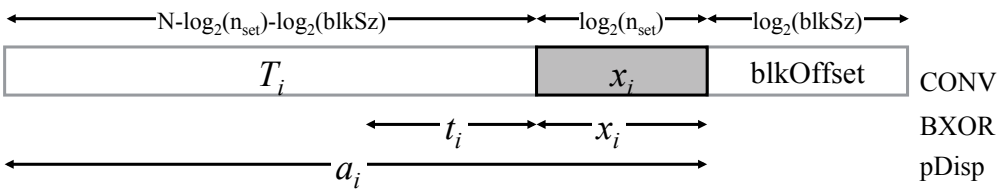


Figure 3.3: Decomposition of a N-bit memory address in various indexing methods. x_i and t_i represent partial index bits of block address a_i.

Even when the *mod* operation is needed, the delay of FUP could still be implemented in no more than 2 cycles. Since GPU is highly optimized for throughput and GPGPU applications are not sensitive to cache latency [4], this delay is easily compensated by reduced intra-warp concentration.

FUP naturally adapts to streaming-like coherent loads where inter-warp feature bits tend to be concentrated in the lower bits of block addresses.

3.2.4 Comparison of Various Indexing Methods

We compare FUP to several presentative indexing methods. Given a N-bit memory address, Figure 3.3 illustrates the decomposition of address bits in cache indexing methods that we will compare with. In the conventional method (*CONV*), the $log_2(n_{set})$ bits in the middle of the block address, i.e., (x_i) bits, are selected as the set index.

Pseudo-random indexing methods are often used to randomize accesses to cache sets. Among them, XOR-based indexing methods are by far the most extensively studied. We choose the bitwise-XOR (*BXOR*) [25] as a representative of pseudo-random indexing methods. BXOR extends the bits for set index calculation via $x_i \oplus t_i$, where t_i also has $log_2(n_{set})$ bits. As we can see from Figure 3.3, both *CONV* and *BXOR* are incapable of covering all feature bits of the intra-warp addresses. The $2 \times log_2(n_{set})$ bits used by *BXOR* mainly cover small strides. For large strides, *BXOR* includes invariable bits for index calculation, leading to high intra-warp concentration. Because of the large range of feature bits, simply altering the bits for *BXOR*, such as the scheme reported in [58], is still unlikely to make *BXOR* adapt to all kinds of strides.

Kharbutli et al. [47] proposed two prime numbers based indexing methods, prime modulo (*pMOD*) and prime displacement (*pDisp*). pMOD uses a prime number of sets in the cache. pDisp

Table 3.1: Baseline GPGPU-Sim Configuration for FUP Study

# of SMs	30 (15 clusters of 2)
SM Configuration	1400Mhz, Reg #: 32K, Shared Memory: 48KB, SIMD Width: 16, warp: 32 threads, max threads per SM: 1024
Caches / SM	Data: 32KB/128B-line/8-way, Constant: 8KB/64B-line/24-way, Texture: 12KB/128B-line/2-way
Branching Handling	PDOM based method [21]
Warp Scheduling	GTO
Interconnect	Butterfly, 1400Mhz, 32B channel width
L2 Unified Cache	768KB, 128B line, 16-way
Min. L2 Latency	120 cycles (compute core clock)
# Memory Partitions	6
# Memory Banks	16 per memory partition
Memory Controller	Out-of-Order (FR-FCFS), max request queue length: 32
GDDR5 Timing	$t_{CL} = 12$, $t_{RP} = 12$, $t_{RC} = 40$, $t_{RAS} = 28$, $t_{RCD} = 12$, $t_{RRD} = 6$, $t_{CDLR} = 5$, $t_{WR} = 12$

calculates the set index as follows: $index = (p \times T_i + x_i)$ **mod** n'_{set}, where tag T_i has $N - log_2(n_{set}) - log_2(blkSz)$, p is a prime number and n'_{set} is the largest prime number that is smaller than n_{set}. The introduction of prime numbers based modulo operation disturbs the high regularity in column-major strided access pattern, dispersing bursty intra-warp accesses into the majority of available cache sets ($n'_{set} < n_{set}$). However, using a fraction of the available cache sets underutilizes cache capacity, and inherently causes intra-warp concentration at the degree of n_{set}/n'_{set}.

3.3 Experimental Evaluation

3.3.1 Experimental Methodology

We use GPGPU-Sim [4] (version 3.2.1), a cycle-accurate simulator, for the performance evaluation of FUP cache indexing method, as discussed in section 3.2. The main characteristics of our baseline GPU architecture are summarized in Table 3.1. We present and discuss the performance impacts of cache indexing methods on highly cache-sensitive GPGPU applications. The highly cache-sensitive benchmarks we study are from Rodinia [12], SHOC [17], PolyBench/GPU [28], and MapReduce [32]. Table 3.2 lists a brief description of each benchmark, cache sensitivity type,

Table 3.2: Highly Cache-Sensitive GPGPU (CUDA) Benchmarks for FUP Study

Abbr.	Application	Suites	Sensitivity	Input
ATAX	matrix-transpose and vector multip.	PolyBench	Associativity	$8K \times 8K$
BICG	kernel of BiCGStab linear solver	PolyBench	Associativity	$8K \times 8K$
MVT	Matrix-vector-product transpose	PolyBench	Associativity	8K
GES	Scalar-vector-matrix multiplication	PolyBench	Associativity	4K
SYRK	Symmetric rank-K operations	PolyBench	Associativity	512×512
SYR2K	Symmetric rank-2K operations	PolyBench	Capacity	256×256
KMN	Kmeans Clustering	Rodinia	Capacity	28k 4x features
BFS	Breadth-First-Search	Rodinia	Capacity	5M edges
SPMV	Sparse matrix multiplication	SHOC	Capacity	default
IIX	Inverted Index	Mars	Capacity	6.8M

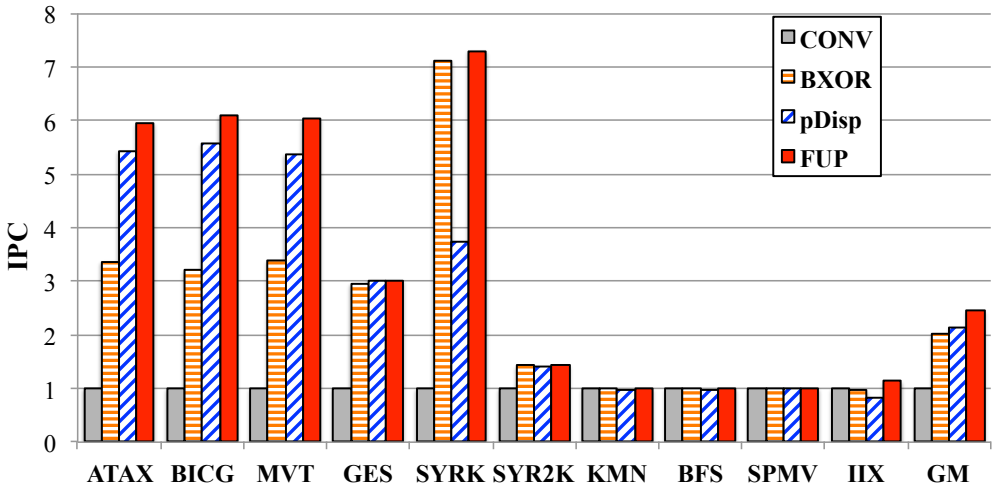


Figure 3.4: The impacts of cache indexing methods on IPC.

and the input sizes that we use to motivate the problem of intra-warp associativity conflicts in Section 2.1 and evaluate the performance of FUP in the rest of the chapter. All of the benchmarks are run to completion which takes between 70 million and 1.5 billion instructions. Among all of the evaluations, *CONV* is taken as the baseline indexing method for comparison.

3.3.2 Instructions Per Cycle (IPC)

Figure 3.4 shows the IPC results of the benchmarks when using different indexing methods. On average, *BXOR* [25], *pDisp* [47], and FUP outperform *CONV* by 2.01X, 2.14X, and 2.46X, respectively. For associativity-sensitive benchmarks, i.e., ATAX, BICG, MVT, GES, SYRK and

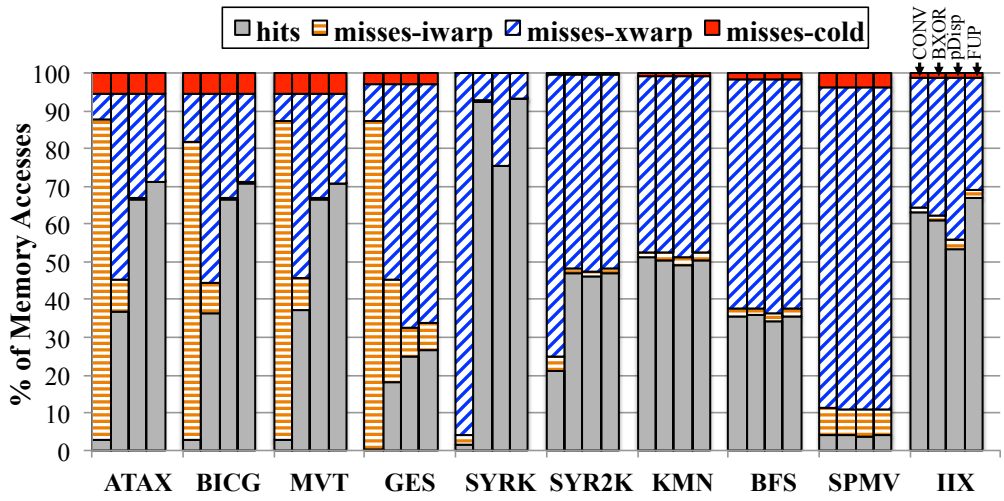


Figure 3.5: The misses/hits of cache accesses when various cache indexing methods are applied.

SYR2K, the three methods achieve a performance improvement of 3.21X, 3.70X and 4.36X, respectively. Across all the benchmarks, *BXOR* and *FUP* do not hurt performance, but *pDisp* downgrades the performance of IIX by 18.4%. This performance degradation in IIX is mainly because *pDisp* underutilizes cache capacity and constantly introduces an intra-warp concentration of 32/31 that could have been eliminated by CONV.

3.3.3 Cache Hits and Misses

Figure 3.5 shows the results of cache hits and misses when using different indexing methods. In general, the IPC performance directly comes from the reduced intra-warp associativity conflicts. In the baseline, *CONV* constantly incurs high intra-warp misses in ATAX, BICG, MVT, and GES. By converting associativity contention into capacity contention, *BXOR* successfully reduces intra-warp misses so that cache hit rates increase. Meanwhile, *pDisp* and *FUP* eliminate the intra-warp misses in these benchmarks and cache hit rates further increase, demonstrating their superior performance over *BXOR*. *FUP* outperforms *pDisp* in preserving cache locality because of the utilization of all available cache sets. In SYRK and SYR2K, *CONV* maps inter-warp accesses into the same cache sets, while *BXOR, pDisp,* and *FUP* reduce the inter-warp concentration

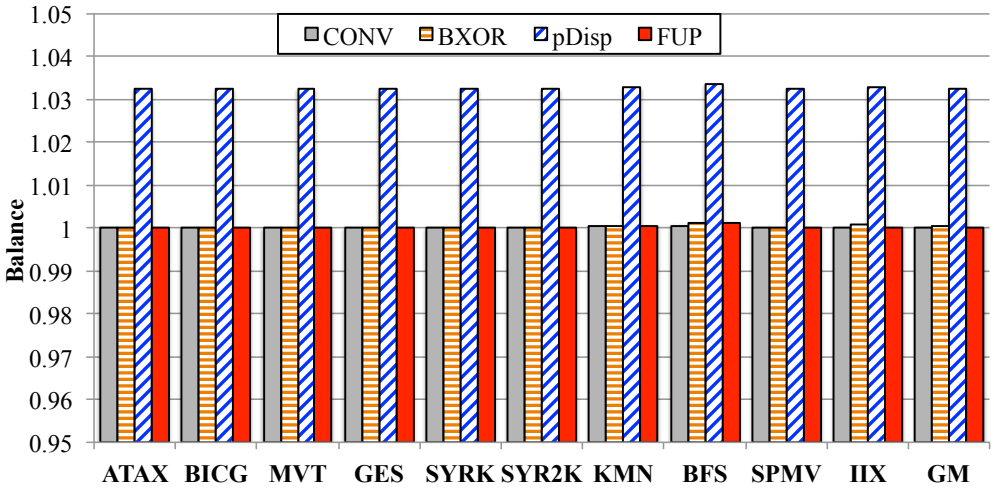


Figure 3.6: The balance of cache access distribution in different cache indexing methods.

by widely spreading these inter-warp accesses. For other benchmarks, cache indexing methods introduce small disturbances into cache hit/miss rates that are reflected in their IPC variance.

3.3.4 Balance

Balance quantifies the accumulated uniformity of distributing the addresses over all the sets in the cache and can be measured using following equation [47]:

$$balance = \frac{\sum_{j=1}^{n_{set}} \frac{b_j \times (b_j+1)}{2}}{\frac{m}{2 \times n_{set}} \times (m + 2 \times n_{set} - 1)} \tag{3.2}$$

where b_j is the total accesses to j^{th} set and m is the total cache accesses. The per-set accesses are weighted by $\frac{b_j \times (b_j+1)}{2}$, and the denominator of the equation gives the sum of the weights of all sets in a perfectly random address distribution. A lower balance value indicates a better address distribution over all sets, and a value of 1 indicates an ideal distribution. As shown in Figure 3.6, all methods except *pDisp* eventually accumulate an even distribution for all L1D accesses across all benchmarks. The 3.2% average change of balance in *pDisp* comes from the fact that it uses only 31 out of 32 sets in the baseline L1D. More importantly, this metric is disconnected with the actual performance improvement as shown in Figure 3.4, because it quantifies the final distribution

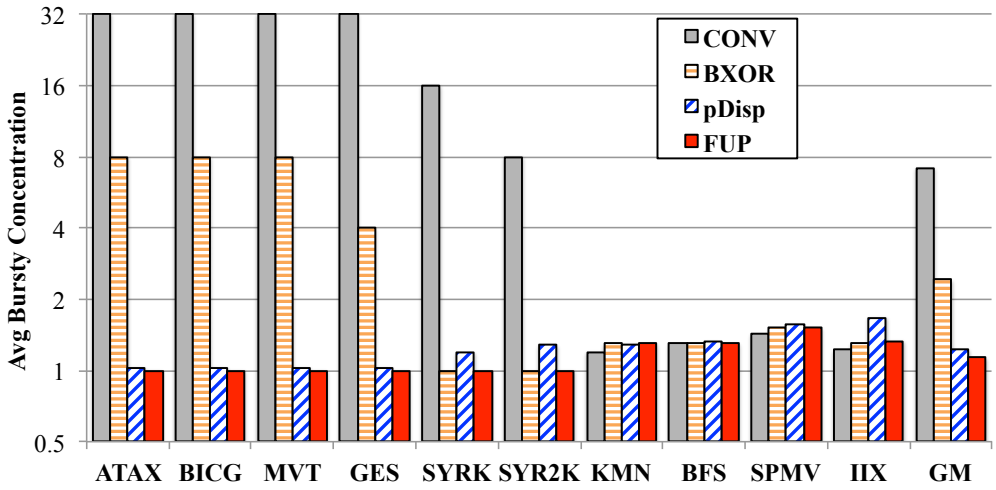


Figure 3.7: Average intra-warp concentration in different cache indexing methods. The y-axis is in logarithmic scale.

of cache accesses. Thus, balance is incapable of describing the dynamic intra-warp contention on cache associativity when intra-warp accesses are pathologically concentrated on a few sets.

3.3.5 Average Intra-warp Concentration

Figure 3.7 presents average bursty concentration when using different indexing methods. For the associativity-sensitive benchmarks, *CONV* constantly maps the intra-warp accesses of ATAX, BICG, MVT and GES into a single set (an average bursty concentration of 32) and that of SYRK and SYR2K into 2 and 4 sets, respectively. Such degree of intra-warp concentration causes high intra-warp misses in ATAX, BICG, MVT and GES, and inter-warp misses in SYRK and SYR2K. By taking more bits of the memory address into consideration, *BXOR* achieves ideal bursty concentrations in SYRK and SYR2K that have medium strides, but still causes high bursty concentrations in ATAX, BICG, MVT, and GES that have large strides. By keeping intra-warp concentration within the size of cache associativity, *BXOR* converts the majority of intra-warp misses in *CONV* into inter-warp misses, which is the key to improving performance. Meanwhile, *pDisp* incurs an average bursty concentration of 1.09 for the associativity-sensitive benchmarks, slightly drifting away from its ideal intra-warp concentration of 1.03 (=32/31). By spreading intra-warp accesses into most of the addressable sets, *pDisp* not only avoids intra-warp contention, but also provides

better potential in L1D locality preservation. This advantage explains 15.3% IPC improvement of *pDisp* over *BXOR* in associativity-sensitive benchmarks, as shown in Figure 3.4. Other than IIX, *pDisp* also incurs higher intra-warp concentration in SYRK and SYR2K that have small strides. For the other benchmarks, *BXOR* and *pDisp* have similar results. By taking all feature bits into account in the cache index calculation, *FUP* constantly adapts to both large and small strides, and therefore achieves a perfect intra-warp concentration in all associativity-sensitive benchmarks. For all the other benchmarks, *FUP* incurs slightly larger intra-warp concentration than *CONV*, which is absorbed by the massive parallelism so that IPC is barely impacted. The variance of intra-warp concentrations in *BXOR, pDisp,* and *FUP*, closely matches their IPC improvement, which suggests the effectiveness of intra-warp concentration as a metric to evaluate the quality of conflict management for GPU cache indexing methods.

3.4 Related Work

Since we have compared bitwise-XOR and prime displacement methods in Section 3.2.4 in detail, we briefly review other related work in this section. Pseudo-random cache indexing methods have been extensively studied to reduce conflict misses. Topham et al. [86] used XOR to build a conflict-avoiding cache; Seznec and Bodin [74, 7, 76] combined XOR indexing and circular shift in a skewed associative cache to form a perfect shuffle across all cache banks. XOR is also widely used for memory indexing [66, 48, 67, 35, 19, 102]; however, no work has exploited the pathological behaviors of cache indexing methods in GPUs.

Another common approach to reduce conflict misses is to use a secondary indexing method for alternative cache sets when conflicts happen. This category of work includes skewed-associative cache [74], column-associative cache [2], and v-way cache [65]. Even though FUP is designed for LRU caches, we believe *FUP* is orthogonal to these cache architectures and could be combined with them into GPUs to further reduce conflict misses.

Some works have also noticed that certain bits in the address are more critical in reducing cache miss rate. Givargis [24] used offline profiling to detect feature bits for embedded systems.

This scheme is only applicable for embedded systems where workloads are often known prior to execution. Ros et al. [71] proposed ASCIB, a three-phase algorithm, to track the changes in address bits at runtime and dynamically discard the invariable bits for cache indexing. ASCIB needs to flush certain cache sets whenever the cache indexing method changes, so it is best suited for direct-mapped cache. ASCIB also needs extra storage to track the changes in the address bits. *FUP* proactively covers all the feature bits in the intra- and inter-warp addresses to realize perfect intra-warp concentration in strided access patterns and incurs no storage overhead.

Regarding the problem of intra-warp conflict misses in GPU architecture, MRPB [40] is the most related work. Instead of increasing the utilization of L1D, MRPB attempts to reorder/prioritize per-warp accesses and aggressively bypass L1D when intra-warp conflicts stall the LD/ST unit. *FUP* solves the problem by spreading intra-warp accesses over all cache sets. Without any storage overhead or complicated logic for request reordering and prioritization, *FUP* opens another avenue to solve the problem of intra-warp conflicts in GPUs.

3.5 Summary

The inclusion of on-chip data caches into GPU was intended to reduce memory operation latency and save bandwidth. However, the high thread counts often destroy the locality in L1D and cause high intra-warp associativity conflicts upon memory divergence. Without spreading intra-warp accesses into all available sets, associativity conflicts serialize the LD/ST unit and undercut the potential of applying other optimization. Thus, it is desirable to investigate how GPU cache indexing methods should be tailored to disperse bursty intra-warp accesses and eliminate conflicts among them.

In this work, we first defined a metric called intra-warp concentration and a type of characteristics called feature bits to guide the design of GPU cache indexing method. In addition to the metric and feature bits, we proposed a *Full-Permutation (FUP)* based GPU cache indexing method that uses all feature bits to calculate the set index via two-level XOR gates. By adopting FUP, 10

highly cache-sensitive benchmarks experience an average 2.46× performance improvement, outperforming the two state-of-the-art methods, *BXOR* and *pDisp*, by 22% and 15%, respectively. Meanwhile, intra-warp concentration is also proven to be an effective metric to quantify the dynamic uniformity of intra-warp accesses over all cache sets. With the help of FUP, GPUs can adapt to any stride size that does not overflow device memory capacity.

Chapter 4

DaCache: Memory Divergence-Aware GPU Cache Management

4.1 Introduction

GPUs allow an application to be programmed as thousands of threads running the same code in a lock-step manner, in which warps of 32 threads can be scheduled for execution in every cycle with zero switching overhead. The massive parallelism from these Single-Instruction Multiple Data (SIMD) threads helps GPUs achieve a dramatic improvement in computational power compared to CPUs. To reduce the latency of memory operations, recent GPUs have employed multiple levels of data caches to save off-chip memory bandwidth when there is locality within the accesses.

Due to massive multithreading, per-thread data cache capacity often diminishes. For example, Fermi supports a maximum of 48 warps (1536 threads) on each Streaming Multiprocessor (SM), and these warps share 16KB or 48KB L1 Data Cache (L1D) [59]. Thus, coalescing each warp's per-thread global memory accesses into fewer memory transactions not only minimizes the consumption of memory bandwidth, but also alleviates cache contention. When a warp's accesses cannot be coalesced into one or two cache blocks, which is referred to as memory divergence, its cache footprint is often boosted by one order of magnitude, e.g., from 1 to 32 cache blocks. Such an explosive increase in per-warp cache footprint leads to severe contention among warps, i.e., inter-warp contention, on limited L1D capacity.

Under the lock-step execution model, a warp is not ready for execution until all of its threads are ready (e.g., no thread has outstanding memory request). Meanwhile, cache-sensitive GPGPU workloads often have high intra-warp locality [69, 70], which means data blocks are re-referenced by their fetching warps. Intra-warp locality is often associated with strided accesses [40, 90]; however, as described in Section 2.1.2, strided accesses lead to divergent memory accesses when stride size is large. The execution model, intra-warp locality, and potential memory divergence

together pose a great challenge for GPU cache management, i.e., data blocks fetched by a divergent load instruction should be cached as a wholistic group. Otherwise, a warp is not ready for issuance when its blocks are partially cached. This challenge demands a GPU-specific cache management that can resist inter-warp contention and minimize partial caching. Though there are many works on thrashing-resistant cache management for multicore systems [64, 22, 38, 46], they are all divergence-oblivious, i.e., they make caching decisions at the per-thread access level rather than at the per-warp instruction level.

Recently, GPU warp scheduling has been studied to alleviate inter-warp contention from its source. Several warp scheduling techniques have been proposed based on various heuristics. For example, CCWS [69], DAWS [70], and CBWT [15] rely on detected L1D locality loss, aggregated cache footprint, and varying on-chip network latencies, respectively, to throttle concurrency at runtime. Limiting the number of actively scheduled warps directly reduces inter-warp contention and delivers higher reductions of cache misses than the Belady [6] replacement algorithm in highly cache-sensitive GPGPU benchmarks [69]. We observe that coherent loads may also carry high intra- and inter-warp locality, but are vulnerable to the thrashing from both inter- and intra-warp divergent loads. However, warp scheduling can only be exploited to alleviate inter-warp contention at a coarse granularity, i.e., warp level. Thus, there is still a need of a salient cache mechanism that can manage L1D locality at both levels and, more importantly, sustain a good supply of Fully Cached Warps (FCW) to keep warp schedulers busy.

Taken together, for a greater good on reducing cache misses and maximizing the occupancy of GPU cores, it is imperative to integrate warp scheduling with the GPU-specific cache management for a combined scheme that can overcome the inefficiency of existing GPU caches. To this end, we present a Divergence-Aware Cache (DaCache) management scheme to mitigate the impacts of memory divergence on L1D locality preservation. Based on the observation that warp scheduling shapes the locality pattern inside L1D access stream, DaCache gauges insertion positions of incoming data blocks according to the fetching warp's scheduling priority. Specifically, new blocks are inserted into L1D in an orderly manner based on their issuing warps' scheduling priorities.

DaCache also prioritizes coherent loads over divergent loads in insertion to alleviate intra-warp contention. In addition, cache ways are conceptually partitioned into two regions, locality region and thrashing region, and replacement candidates are constrained within thrashing region to increase thrashing resistance. If no replacement candidate is available in thrashing region, L1D bypassing is enabled. We propose a simple mechanism to dynamically adjust the partitioning. All these features in our DaCache design need simple modifications to existing LRU caches.

In summary, this chapter makes the following contributions:

- Evaluating caching effectiveness of GPU data caches for both memory-coherent and memory-divergent GPGPU benchmarks, and present the problem of partial caching in existing GPU cache management;
- Proposing a Divergence-Aware Cache management technique, namely DaCache, to orchestrate warp scheduling and cache management for GPGPUs. By taking prioritization logic of warp scheduling into cache management, thrashing traffic can be quickly removed so that cache blocks of the most prioritized warps can be fully cached in L1D; in turn the increased number of fully cached loads provides more ready warps for warp schedulers to execute;
- Designing a dynamic partitioning algorithm in DaCache to increase thrashing resistance and implement it in a cycle-accurate simulator. Experimental results show that it can improve caching effectiveness and improve the performance by 40.4% over baseline GPU architecture, outperform two thrashing resistance cache management, RRIP and DIP, by 40% and 24.9%, respectively.

The rest of chapter is organized as follows: Section 4.2 details the design of DaCache; Experimental results and related work are presented in Section 4.3 and Section 4.4, respectively. Section 4.5 summarizes this chapter.

4.2 Divergence-Aware GPU Cache Management

As described in the Section 2.2, divergent load instructions lead to severe cache misses in L1D, especially inter-warp capacity conflict misses. With more data blocks not being found in

L1D, the number of warps that can be actively scheduled are significantly reduced. To address this problem, we propose *Divergence-Aware Cache (DaCache)* management for GPUs. Based on the observation that the re-reference interval of cache blocks are shaped by warp schedulers, DaCache aims to exploit the prioritization information of warp scheduling logic, protect the cache blocks of high-priority warps from being evicted by the blocks of low-priority warps, and reduce the problem of partial caching as defined in Section 2.2. In doing so, DaCache can alleviate conflict misses across concurrent warps such that more warps can locate all data blocks from L1D for their load instructions. We refer to such warps as Fully Cached Warps (FCWs).

4.2.1 High-level Description of DaCache

Figure 4.1 shows a conceptual idea of DaCache in maximizing the number of FCWs. In this example, we assume four warps concurrently execute a for-loop body that has one divergent load instruction. At runtime, each warp generates four cache accesses in each loop iteration, and the fetched cache blocks are re-referenced across iterations. This program example is from the strided access pattern in our evaluated CUDA benchmarks. Ideally, all loads can hit in L1D due to high intra-warp locality. Severe cache contention caused by massive parallelism and scarce L1D capacity can easily thrash the locality in L1D. In order to resist thrashing, a divergence-oblivious cache management may fairly treat accesses from all warps, leading to the scenario that all warps miss one block in current iteration. By taking warp scheduling prioritization and memory divergence into consideration, DaCache aims at cache misses concentrated at warps that have lower scheduling priorities, such as *W3* and *W4*. Consequently, warps with higher scheduling priorities, such as *W1* and *W2*, can be fully cached so that they are immediately ready to execute the next iteration of the for-loop body.

DaCache leverages both warp scheduling-awareness and memory divergence-awareness to maximize the number of FCWs. This goal necessitates several innovative changes on GPU cache management policies. In general, cache management consists of three components: replacement, insertion, and promotion policies [99]. *Replacement policy* decides which block in a set should be

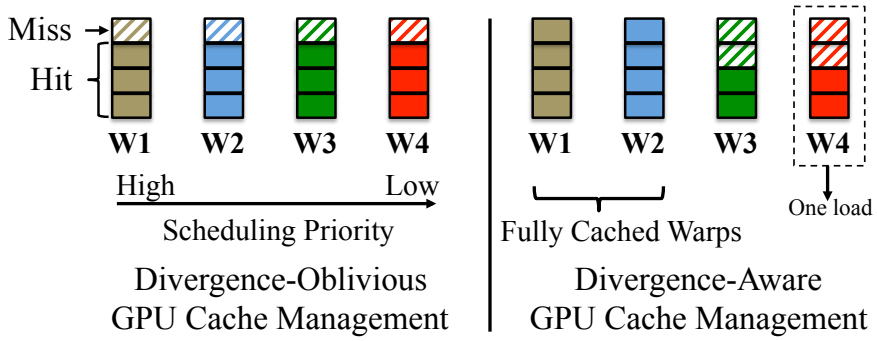


Figure 4.1: A conceptual example comparing the consequences of divergence-oblivious and divergence-aware cache management. Divergence-aware cache management can fully cache more warps with high scheduling priorities.

evicted upon a conflicting cache access, *insertion policy* defines a new block's replacement priority, and *promotion policy* determines how to update the replacement priority of a re-referenced block. For example, in LRU caches, blocks at the LRU position are immediate replacing candidates; new blocks are inserted into the MRU position of the LRU chain; re-referenced blocks are promoted to the MRU position.

4.2.2 Gauged Insertion

In conventional LRU caches, since the replacement candidates are always selected from the LRU ends, blocks in the LRU-chains have different lifetimes to stay in cache. For example, blocks at the MRU ends have the longest lifetime, while blocks at LRU ends have shortest lifetime. Based on this characteristic, locality of L1D blocks can be differentially preserved by inserting blocks at different positions in the LRU-chains according to their re-reference intervals. For example, blocks can be inserted into MRU, central, and LRU positions if they will be re-referenced in the immediate, near, and distant future, respectively. However, it is challenging for GPU caches to predict re-reference intervals of individual cache blocks from the thrashing-prone cache access streams.

Since there is often high intra-warp data locality among memory-divergent GPGPU benchmarks, the cache blocks of frequently scheduled warps have short re-reference intervals, while the

blocks of infrequently warps have long re-reference intervals. Under GTO warp scheduling, old warps are prioritized over young warps and are more frequently scheduled. Thus, we can use each warp's GTO scheduling priority to predict its blocks' reference intervals. Based on this observation, the insertion position (*way*) in DaCache is gauged as:

$$way = min\{W_{Prio} \times N_{Sched} \times Width/N_{Set}, Asso - 1\} \tag{4.1}$$

where W_{Prio} is the issuing warp's scheduling priority, N_{Sched} is the number of warp schedulers in each SM, N_{Set} is the number of cache sets in L1D, $Width$ is the SIMD width, and $Asso$ is the cache associativity. Behind this gauged insertion policy, we assume the accesses from divergent loads (up-to $Width$ accesses) are equally distributed into N_{set} sets, and $Width/N_{Set}$ quantifies average intra-warp concentration in each cache set. Since L1D is shared by N_{Sched} warp schedulers, warps with the same priority but from different warp schedulers are assigned with the same insertion positions. Thus, the cache blocks of consecutive warps from the same warp scheduler are dispersed by $N_{Sched} \times Width/N_{Set}$. For example, in our baseline GPU (2 warp schedulers per SM; 32 threads per warp; 32 sets per L1D), two warps with priorities of 0 and 2 are assigned insertion positions of 0 and 4, respectively. The gauged insertion policy is illustrated in Figure 4.2. In the figure, data blocks of "oldest warp", "median warp", and "youngest warp" are initially inserted into the MRU, central, and LRU positions, respectively. At runtime, the majority of the active warps are infrequently scheduled and share the LRU insertion position. By doing so, blocks are inserted in the LRU-chain in an orderly manner based on their issuing warps' scheduling priorities.

GPU programs often have a mix of coherent and divergent loads, which are assigned with the same insertion positions under the gauged insertion policy. Consequently, coherent loads will be interleaved with divergent loads. Interleaved insertion can make coherent loads vulnerable to thrashing from the bursty behaviors of divergent loads. The thrashing to coherent loads may not be limited to inter-warp contention. Figure 2.4 demonstrates the existence of intra-warp conflict misses in conventional LRU caches. We propose to explicitly prioritize coherent loads over divergent loads by inserting blocks of coherent loads into MRU positions, regardless of their issuing warps' scheduling priorities. Coherent loads may not carry any locality, and inserting their blocks

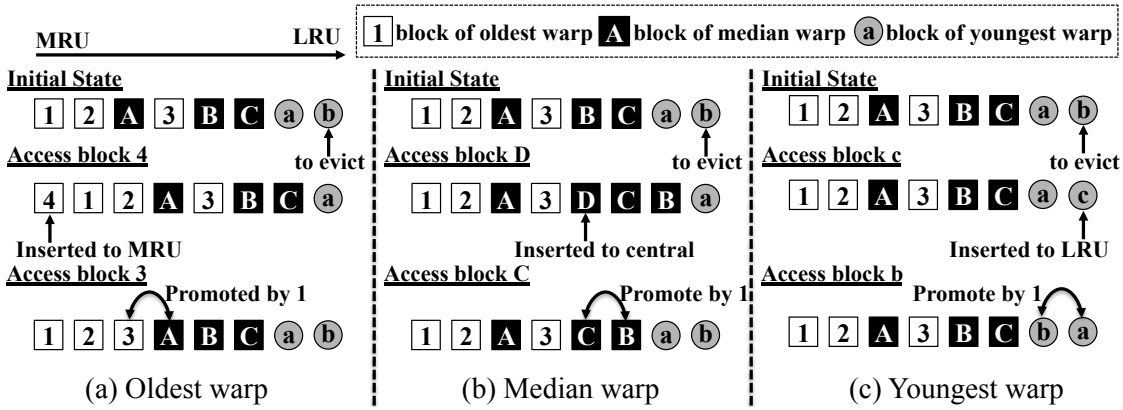


Figure 4.2: Illustrative example of insertion and promotion policies of DaCache.

into MRU positions is adversary to locality preservation. We use a victim cache to detect whether coherent loads have intra-warp locality, and then MRU insertion and LRU insertion are applied to coherent loads with and without locality, respectively. Motivated by the observation from Figure 2.3b, we empirically use MRU insertion for divergent load instructions with no more than 5 memory requests.

Each entry of the victim cache has two fields, PC and data block tag. For a 48bit virtual address space, maximally the PC field needs 45 bits and the tag field needs 41 bits. Since only the mostly prioritized warp is sampled at runtime to detect the locality information of coherent loads, a 16-entry victim cache is sufficient across the evaluated benchmarks, which incurs only 172B storage overhead on each SM. The dynamic locality information of each coherent load is stored in a structure named Coherent Load Profiler (CLP). CLP entries have two fields, PC field (45 bits) and one flag field (1 bit) to indicate locality information. A 32-entry CLP incurs 184B storage overhead. Note that, when a load instruction is issued into LD/ST, memory access coalescing in MACU and CLP lookup can be executed in parallel. Once the locality information of a coherent load is determined, victim cache can be bypassed to avoid repetitive detection. Such storage overhead can be eliminated by embedding the potential locality information into PTX instructions via compiler support. We leave this as our future work.

Note that the insertion policy only gives an initial data layout in L1D to approximate re-reference intervals. During the runtime, the initial data layout can be easily disturbed because re-referenced blocks are directly promoted to the MRU positions, regardless of their current positions in the LRU-chain. In other words, this MRU promotion can invert the intention of DaCache insertion policy. Partially motivated by the incremental promotion in PIPP [99] that promotes re-referenced block by 1 position along the LRU-chain, DaCache also adopts a fine-grained promotion policy to cooperate with the insertion policy. Figure 4.2 illustrates a promotion granularity of 2 positions. Our experiments in Section 4.3.7 show that a promotion granularity of 4 achieves the best performance for the benchmarks we have evaluated.

4.2.3 Constrained Replacement

In general, in LRU caches, the block on the LRU end is considered as the replacement candidate. However, as we model cache contention by allocating cache block on miss and reserving blocks for outstanding requests [4], the block at the LRU position may not be replaceable. A replaceable block that is the closest to the LRU position is then selected. Thus, the replacement decision is no longer constrained on the LRU end, and any block in the set may be a replacement candidate. Such unconstrained replacement positions make inter-warp cache conflicts very unpredictable.

To protect the intention of gauged insertion, we introduce a constrained replacement policy in DaCache so that only a few blocks close to the LRU end can be replaced. This constrained replacement conceptually partitions the cache ways into two portions, locality region and thrashing region. Replacement can then only be made inside the thrashing region. This partitioning (p) can be calculated as: $p = \frac{Asso \times F}{SIMD\_Width/N_{Set}} - 1$, where F is a tuning parameter in the range between 0 and 1. Denoting the MRU and LRU ends with the way indexes of 0 and *Asso-1*, respectively, the locality region is located in the range from the 0^{th} to the p^{th} way of a cache set, while the thrashing region occupies the other ways. We tune the value of F to have the optimal static partitioning p. Besides, all sets in each L1D are equally partitioned.

Given the gauged insertion policy, this logical partitioning of L1D accordingly divides all active warps into two groups, locality warps and thrashing warps. If a warp's scheduling priority is higher than $(p+1)/N_{Sched}$, it's a thrashing warp, otherwise it is considered as a locality warp. The cache blocks of locality warps are inserted into the locality region using the gauged insertion policy so that they can be less vulnerable to thrashing traffic. By doing so, locality warps have a better chance to be fully cached and immediately ready for re-scheduling. In order to cooperate with such a constrained replacement policy, divergent loads of thrashing warps are exclusively inserted into LRU positions so that they can not pollute existing cache blocks in L1D. Though the 3 oldest warps managed by each warp scheduler are mostly scheduled as shown in Figure 2.3, i.e., p=5 in our baseline, our experiments in Section 4.3.4 show that maintaining 2 FCWs per warp scheduler (p=3) actually achieves the optimal performance with the extended insertion and unconstrained replacement policies.

With the constrained replacement policy, replacement candidates may not always be available. Thus, we discuss two complementary approaches to enforce constrained replacement. The first approach is called **Constrained Replacement with L1D Stalling**. It's possible that a replacement candidate cannot be located within our baseline cache model, though at a very low frequency. Once this happens, the cache controller repetitively replays the missing access until one block in the thrashing region becomes replaceable. Stalling L1D is the default functionality within our cache model and then can be used with constrained replacement at no extra cost.

The second approach is called **Constrained Replacement with L1D Bypassing**. Instead of waiting for reallocating reserved cache blocks, bypassing L1D proactively forwards the thrashing traffic into lower memory hierarchy. Without touching L1D, bypassing can avoid not only L1D thrashing, but also memory pipeline stalls. When a bypassed request is back, its data is directly written to register file rather than a pre-allocated cache block [40]. In our baseline architecture, caching in L1D forces the size of missed memory requests to be cache block size. For each cache access of a divergent load instruction, only a small segment of the cache block are actually used, depending on the data size and access pattern. Without caching, the extra data in the cache block

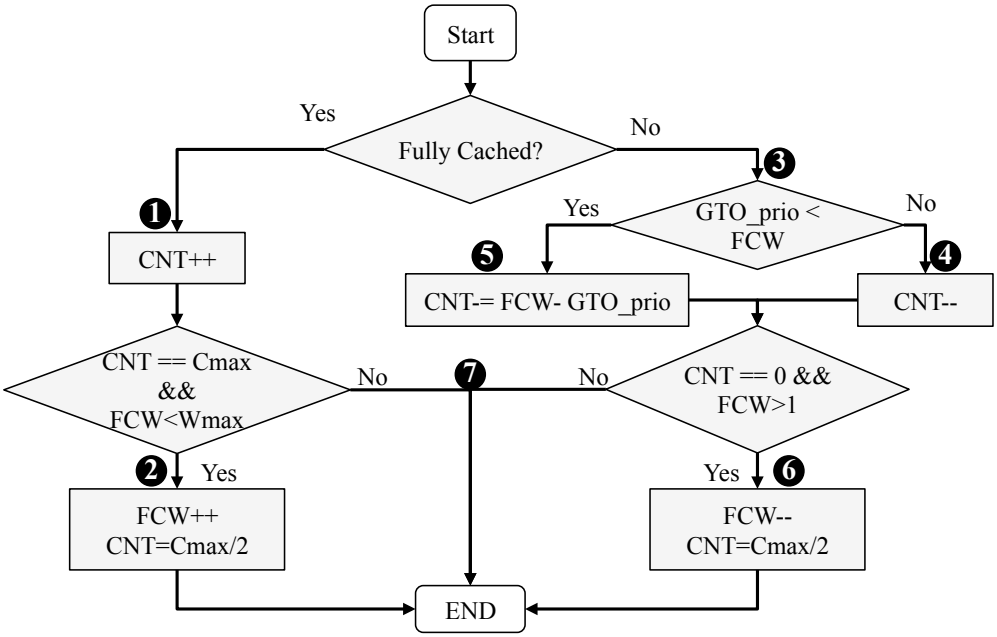


Figure 4.3: Flow of the proposed dynamic partitioning algorithm. *Fully Cached Warps (FCW)* is based on the number of fully cached loads (*CNT*) and each warp's GTO scheduling priority (GTO_prio).

is a pure waste of memory bandwidth. Thus, bypassed memory requests are further reduced to aligned 32B segments, which is the minimum coalesced segment size as discussed in [61].

4.2.4 Dynamic Partitioning of Warps

Our insertion and replacement policies rely on a static partitioning p, which incorporates the scheduling priorities of active warps into the cache management. However, the static choice of p is not very suitable in two important scenarios. First, branch divergence reduces per-warp cache footprint so that the locality region is capable of accommodating more warps. It can be observed from Figure 2.3b that branch divergence enables more warps be actively scheduled. Second, kernels may have multiple divergent load instructions so that the capacity of locality region is only enough to cache one warp from each warp scheduler. For example, SYR2, GES, and SPMV have two divergent loads, while IIX and PVC have multiple divergent loads.

Thus, we propose a mechanism for dynamic partitioning of warps based on the accumulated statistics of fully cached divergent loads. Figure 4.3 shows the flow of dynamically adjusting *Fully Cached Warps (FCW)* based on the accumulated number of fully cached loads (*CNT*) and each warp's GTO scheduling priority (GTO_prio). At runtime, CNT is increased by 1 (❶) for each fully cached load. When CNT is saturated (CNT==Cmax), if FCW has not reached its maximum value (*Wmax*), FCW is increased by 1 and accordingly CNT is reset as *Cmax/2* to track fully cached divergent loads under the new partitioning (❷). For partially cached loads (❸), CNT is decreased differently depending on the issuing warp's scheduling priority. For instance, if a warp's scheduling priority is lower than FCW, CNT is decreased by 1 (❹); otherwise, CNT is decreased by FCW-GTO_prio (❺) to speed up the process of achieving the optimal FCW. When CNT reaches zero, FCW is decreased by 1 so that less warps are assigned into the locality region (❻). In our proposal, each warp scheduler has at least 1 warp in the locality region; while *Wmax* is equal to 48, which is the number of physical warps on each SM. Thus, in the corner cases when FCW is 1 or Wmax (❼), CNT will not be overflowed if it's saturated.

In order to implement the logic of dynamic partitioning, we first use one register (*Div-reg*) to mark whether a load is divergent or not, depending on the number of coalesced memory requests. *Div-reg* is populated when a new load instruction is serviced by L1D. We then use another register (*FCW-reg*) to track whether a load is fully cached or not. *FCW-reg* is reset when L1D starts to service a new load, and is set when a cache miss happens. When all the accesses of the load are serviced, *FCW-reg* being unset indicates a fully cached load. The logic of dynamic partitioning is triggered when a divergent load retires from the memory stage. We empirically use a 8-bit counter for CNT so that it can maximally record 256 consecutive occurrence of fully/partially cached loads, i.e., Cmax=256 in Figure 4.3. CNT is initialized as 128 while FCW is 4. This initial value of FCW is based on our experiments of static partitioning schemes showing that maintaining two FCWs for each warp scheduler has the best overall performance.

Table 4.1: Baseline GPGPU-Sim Configuration for DaCache Study

# of SMs	30 (15 clusters of 2)
SM Configuration	1400Mhz, Reg #: 32K, Shared Memory: 48KB, SIMD Width: 16, warp: 32 threads, max threads per SM: 1024
Caches / SM	Data: 32KB/128B-line/8-way, Constant: 8KB/64B-line/24-way, Texture: 12KB/128B-line/2-way
Branching Handling	PDOM based method [21]
Warp Scheduling	GTO
Interconnect	Butterfly, 1400Mhz, 32B channel width
L2 Unified Cache	768KB, 128B line, 16-way
Min. L2 Latency	120 cycles (compute core clock)
Cache Indexing	Pseudo-Random Hashing Function [58]
# Memory Partitions	6
# Memory Banks	16 per memory partition
Memory Controller	Out-of-Order (FR-FCFS), max request queue length: 32
GDDR5 Timing	$t_{CL} = 12$, $t_{RP} = 12$, $t_{RC} = 40$, $t_{RAS} = 28$, $t_{RCD} = 12$, $t_{RRD} = 6$, $t_{CDLR} = 5$, $t_{WR} = 12$

4.3 Experimental Evaluation

GPGPU-Sim [4] (version 3.2.1), a cycle-accurate simulator, is used for the performance evaluation of DaCache. The main characteristics of the baseline GPU architecture are summarized in Table 4.1. Jia et al. [40] reported that the default cache indexing method employed by this version of GPGPU-Sim can lead to severe intra-warp conflict misses, so we use the indexing method from real Fermi GPUs, pseudo-random hashing function [58]. This indexing method has been adopted in the latest version of GPGPU-Sim. This change in GPGPU-Sim configuration isolates the impacts of intra-warp associativity conflicts from the motivational studies of partial caching in Section 2.2 and performance evaluation of DaCache, and gives a faithful simulation of real GPU hardware.

20 data intensive benchmarks from Rodinia [12], PolyBench/GPU [28], SHOC [17], and MapReduce [32] are used for the performance evaluation of DaCache. For each benchmark, Table 4.2 lists a brief description, status of branch divergence, and the input size that is used in motivational experiments presented in Section 2.2 and performance evaluations in the rest of this

Table 4.2: GPGPU Benchmarks (CUDA) for DaCache Study

#	Abbr.	Application	Suite	Input	Branch
Memory Divergent Benchmarks					
1	ATAX	matrix-transpose and vector mul.	PolyBench	$8K \times 8K$	N
2	BICG	kernel of BiCGStab linear solver	PolyBench	$8K \times 8K$	N
3	MVT	Matrix-vector-product transpose	PolyBench	8K	N
4	SYR	Symmetric rank-K operations	PolyBench	512×512	N
5	SYR2	Symmetric rank-2K operations	PolyBench	256×256	N
6	GES	Scalar-vector-matrix mul.	PolyBench	4K	N
7	KMN	Kmeans Clustering	Rodinia	28K 4x features	N
8	SC	Stream Cluster	Rodinia	256K points	N
9	BFS	Breadth-First-Search	Rodinia	5M edges	Y
10	SPMV	Sparse matrix mul.	SHOC	default	Y
11	IIX	Inverted Index	Mars	6.8M	Y
12	PVC	Page View Count	Mars	100K	Y
Memory Coherent Benchmarks					
13	2DC	2D Convolution	PolyBench	default	N
14	3DC	3D Convolution	PolyBench	default	N
15	2MM	2 Matrix Multiply	PolyBench	default	N
16	3MM	3 Matrix Multiply	PolyBench	default	N
17	COV	Covariance Computation	PolyBench	default	N
18	COR	Correlation Computation	PolyBench	default	N
19	FD	2D Finite Difference Kernel	PolyBench	default	N
20	GS	Gram-Schmidt Process	PolyBench	default	N

section. Benchmark SC repetitively invokes the same kernel 290 times with the default input size (16K points) until the computation completes. Since simulation is several orders of magnitude slower than real hardware, only two kernel invocations in SC are enabled so that its simulation time is reasonable with larger input size (256K points). All of the other benchmarks, ranging from 70 million to 6.8 billion dynamic instructions, are run to completion. The benchmarks are categorized into memory-divergent and memory-coherent ones, depending on the dynamic divergence of load instructions in these benchmarks. In general, memory-divergent benchmarks are more sensitive to cache capacity than memory-coherent benchmarks. Recent works [69, 70, 40, 90] report that high intra-warp L1D locality exists among these cache-sensitive workloads. In addition, BFS, SPMV, IIX, and PVC also have rich branch divergence.

The following cache management techniques are evaluated:

LRU is the baseline cache management. Without further mentioning, all performance numbers are normalized to LRU.

Dynamic Insertion Policy (DIP) [64] consists of both LRU and MRU insertions. Cache misses are sampled from the sets that are dedicated to LRU and MRU insertions, and a winning insertion policy for all other "follower" sets. This mechanism is referred as set-dueling. In our evaluation, 4 sets are dedicated for each insertion policy and the other 24 sets are managed by the winning policy.

Re-Reference Interval Prediction (RRIP) [38] uses Re-Reference Prediction Values (RRPV) to manage blocks in a set. With an M-bit RRPV-chain, new blocks are predicted with RRPVs of 2^M-1 or 2^M-2, depending on the winning policy from the set-dueling mechanism. We implement RRIP with a Frequency Priority based promotion and a 3-bit RRPV chain.

DaCache consists of gauged insertion and incremental promotion (Section 4.2.2), constrained replacement with L1D bypassing (Section 4.2.3), and dynamic partitioning (Section 4.2.4). By default, DaCache has a promotion granularity of 4 and the locality region starts with hosting 2 warps from each warp scheduler. We evaluate DaCache variants with unconstrained replacement (*DaCache-Uncon*) and constrained replacement with L1D stalling (*DaCache-Stall*) to demonstrate the importance of using warp scheduling to guide cache management.

4.3.1 Instructions Per Cycle (IPC)

Figure 4.4 compares the performance of various cache management techniques for both memory-divergent and memory-coherent benchmarks. For memory-divergent benchmarks, RRIP on average has no IPC improvement. The performance gains of RRIP are balanced out by its loss in ATAX, BICG, MVT, and SYR, which exhibit LRU-friendly accesses patterns under GTO. Because of the intra-warp locality, highly prioritized warps leave large amount of blocks in the locality region that no other warps will re-reference, i.e., dead blocks, after they retire from LD/ST units.

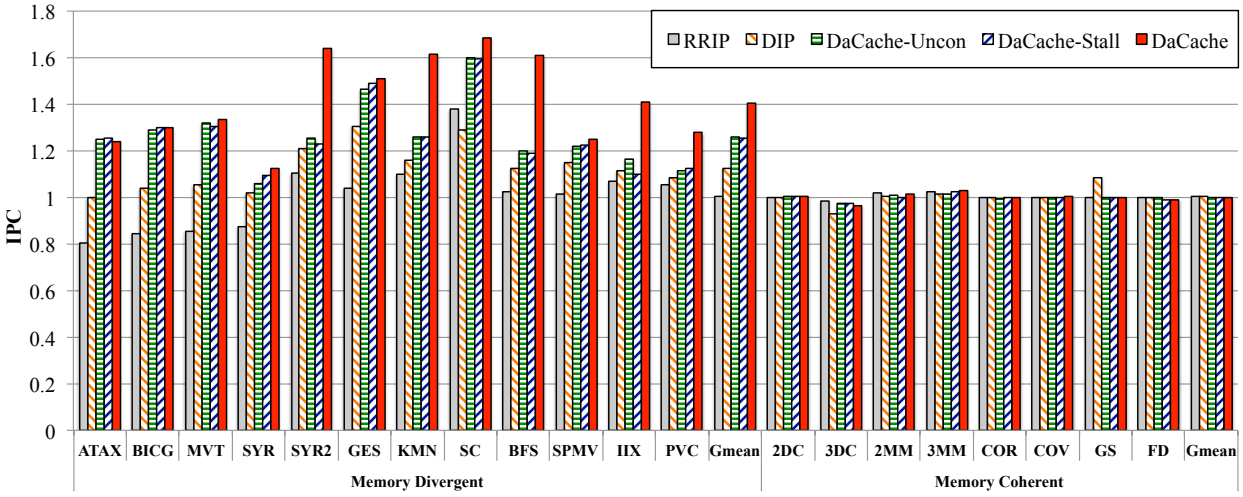
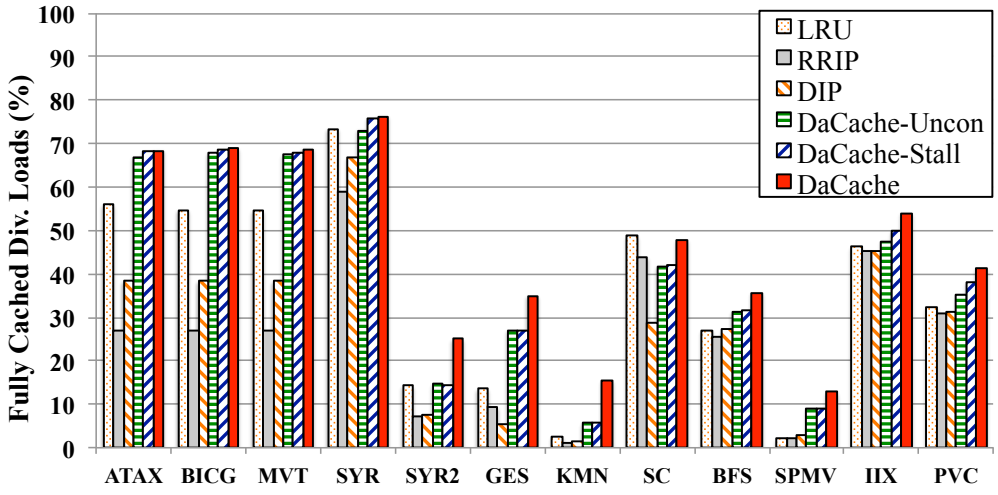


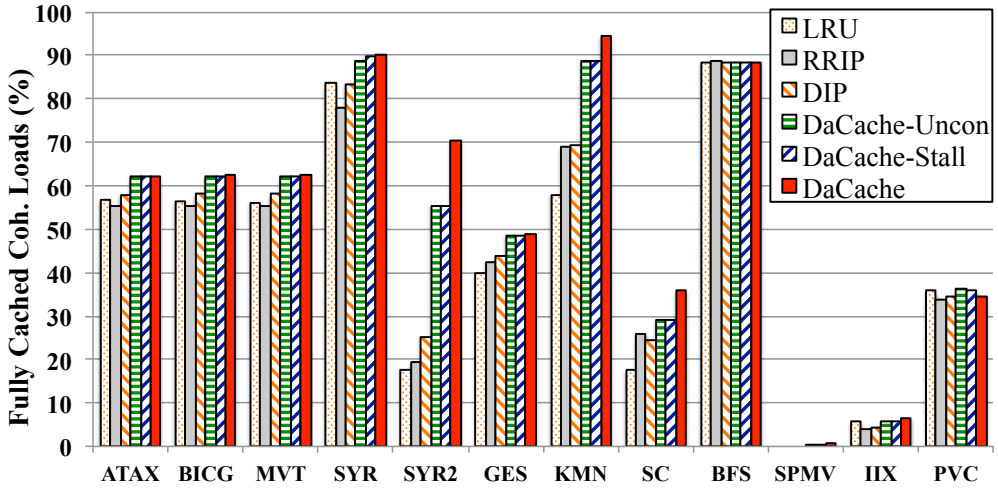
Figure 4.4: IPC of memory-divergent and memory-coherent benchmarks when various cache management techniques are used.

RRIP's asymmetric processes of promotion and replacement make it slow to eliminate the dead blocks, leading to inferior performance in these LRU-friendly benchmarks. Dynamically adjusting between LRU and MRU insertions makes DIP capable of both LRU-friendly and thrashing-prone patterns, therefore DIP has 12.4% IPC improvement. In contrast, *DaCache-Uncon*, *DaCache-Stall*, and *DaCache* have an improvement of 25.9%, 25.6%, and 40.4%, respectively. The performance advantage of *DaCache-Uncon* proves the effectiveness of incorporating warp scheduling into L1D cache management. Based on this warp scheduling-awareness, constrained replacement with L1D stalling (*DaCache-Stall*) has no any extra performance gain. However, enabling constrained replacement with L1D bypassing achieves another improvement of 14.5% in *DaCache*.

Among the memory-coherent benchmarks, only GS has experienced significant performance improvement when DIP is applied. This 8% improvement in DIP is because GS has inter-kernel data locality, and inserting new blocks into LRU position when detected locality is low can help to carry data locality across kernels. We believe this performance improvement will diminish when data size is large enough. For the others, all of the cache management techniques have negligible performance impact. By focusing on memory divergence, DaCache and its variants



(a) Divergent Loads



(b) Coherent Loads

Figure 4.5: Percentages of fully cached load instructions in memory divergent benchmarks.

have no detrimental impacts on memory coherent workloads. We believe DaCache is applicable to a large variety of GPGPU workloads.

4.3.2 Fully Cached Loads

The percentages of fully cached loads (Figure 4.5) explain the performance impacts of various cache management techniques on these memory-divergent benchmarks. As shown in Figure 4.5a, LRU outperforms DIP and RRIP in fully caching divergent loads. Since GTO warp scheduling essentially generates LRU-friendly cache access patterns, LRU cache matches the inherent pattern

so that the blocks of divergent loads are inserted into the contiguous positions of the LRU-chain. In contrast, DIP and RRIP dynamically insert blocks of the same load into different positions of LRU-chain and RRPV-chain, respectively, making it hard to fully cache divergent loads. Thus, the performance impacts of RRIP and DIP mainly come from their capabilities in preserving coherent loads. As shown in Figure 4.5b, for ATAX, BICG, MVT, and SYR, RRIP also achieves less fully cached coherent loads than LRU, therefore it has worse performance than LRU in the four benchmarks; DIP recovers more coherent loads than LRU, but these gains are offset by loss in caching divergent loads, leading to marginal performance improvement. For SYR2, GES, KMN, SC, and BFS, RRIP and DIP improve the effectiveness of caching coherent loads, leading to the performance improvement in the five benchmarks.

DaCache-Uncon, *DaCache-Stall*, and *DaCache* constantly outperform LRU, RRIP, and DIP in fully caching loads, except for benchmark SC. This advantage comes from the following three factors. First, guided by the warp scheduling prioritization, the gauged insertion implicitly enforces LRU-friendliness. Thus, *DaCache-Uncon* achieves 35.1% more fully cached divergent loads. Second, deliberately prioritizing coherent loads over divergent loads alleviates the inter- and intra-warp thrashing from divergent loads. Thus, *DaCache-Uncon* achieves 27.3% more fully cached coherent loads. Third, constrained replacement can effectively improve the caching efficiency for highly prioritized warps. Based on *DaCache-Uncon*, constrained replacement with L1D stalling (*DaCache-Stall*) achieves 37.2% and 27.6% more fully cached divergent and coherent loads than LRU, respectively; while constrained replacement with L1D bypassing (*DaCache*) achieves 70% and 34.1% more fully cached divergent and coherent loads than LRU, respectively. In SC, the divergent loads come from the references to arrays of structs outside of a loop, and references to different members of the struct entry are sequential so that the LRU has the highest percentage of fully cached divergent loads (48.7%). Divergent loads in SC make up only a small portion of the total loads; therefore the number of fully cached coherent loads dominates the performance impacts.

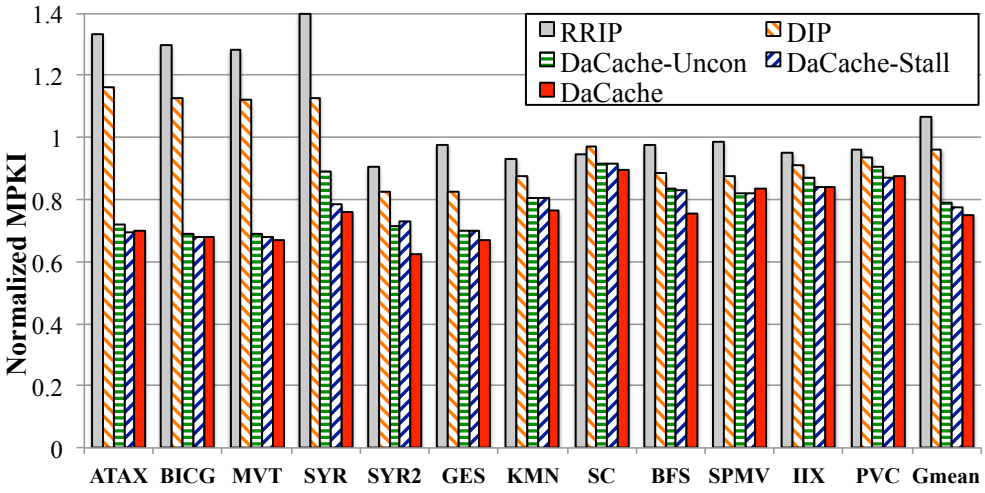


Figure 4.6: MPKI of various cache management techniques.

4.3.3 Misses per Kilo Instructions (MPKI)

We also use MPKI to analyze the performance impacts of various cache management techniques on these memory-divergent benchmarks. As shown in Figure 4.6, except ATAX, BICG, MVT, and SYR, all of the five techniques are effective in reducing MPKIs. Because GPUs are throughput-oriented and rely on the number of fully cached warps to overlap long latency memory accesses, the significant MPKI increase of DIP in the four benchmarks is tolerated so that it doesn't have negative performance impacts. However, RRIP incurs on average a 32.5% increase in MPKIs in the four benchmarks, which leads to 14.5% performance degradation. Across the 12 benchmarks, on average, RRIP increases MPKIs by 6.4%, while DIP reduces MPKIs by 3.8%.

Meanwhile, *DaCache-Uncon*, *DaCache-Stall*, and *DaCache* consistently achieve MPKI reductions. On average, they reduce MPKIs by 20.8%, 22.4%, and 25%, respectively. Though *DaCache-Stall* reduces 1.6% more MPKIs than *DaCache-Uncon*, its potential performance advantage is compromised by adversely inserted L1D stall cycles. On the contrary, bypassing L1D in *DaCache* not only prevents L1D locality from being thrashed by warps with low scheduling priorities, but also enables these thrashing warps to directly access data cached in lower cache hierarchy. So 4.2% more MPKI reductions of *DaCache* brings 40.4% IPC improvement.

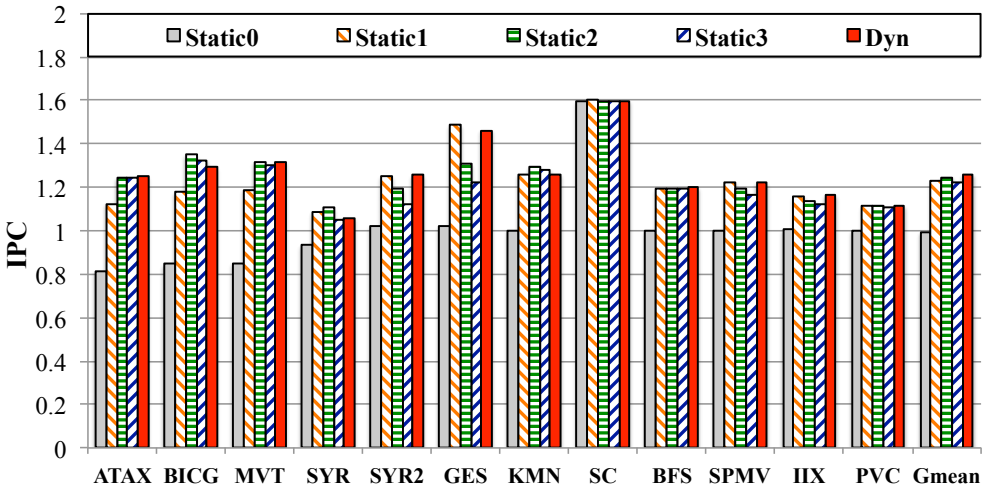


Figure 4.7: DaCache under static and dynamic partitioning.

4.3.4 Static vs. Dynamic Partitioning

Figure 4.7 examines the performance of DaCache when various static partitioning schemes and dynamic partitioning are enabled. For this experiment, the constrained replacement is disabled. *StaticN* means that *N* warps are cached in locality region. For example, in *Static0*, all blocks fetched by divergent loads are initially inserted into the LRU positions. Note that our baseline L1D is 8-way associative, Static3 and Static4 actually lead to identical insertion positions for all warps. Thus, we only compare dynamic partition (*Dyn*) with *Static0*, *Static1*, *Static2*, and *Static3*.

Without any information from warp scheduling, *Static0* blindly inserts all blocks of divergent loads into LRU positions, therefore it becomes impossible to predict which warps' cache block are more likely to be thrashed. On average, this inefficiency of *Static0* incurs 0.1% performance loss. On the contrary, by implicitly protecting 1, 2, and 3 warps for each warp scheduler, *Static1*, *Static2*, and *Static3* achieve performance improvement of 23%, 24.7%, and 21.9%, respectively. Note that *Static2* equally partitions L1D capacity into locality and thrashing regions, and the locality region is sufficient to cache two warps from each warp scheduler. Except IIX and PVC, all other benchmarks have maximally two divergent loads in each kernel, therefore *Static2* has the best performance improvement. Our dynamic partitioning scheme (*Dyn*) achieves a performance

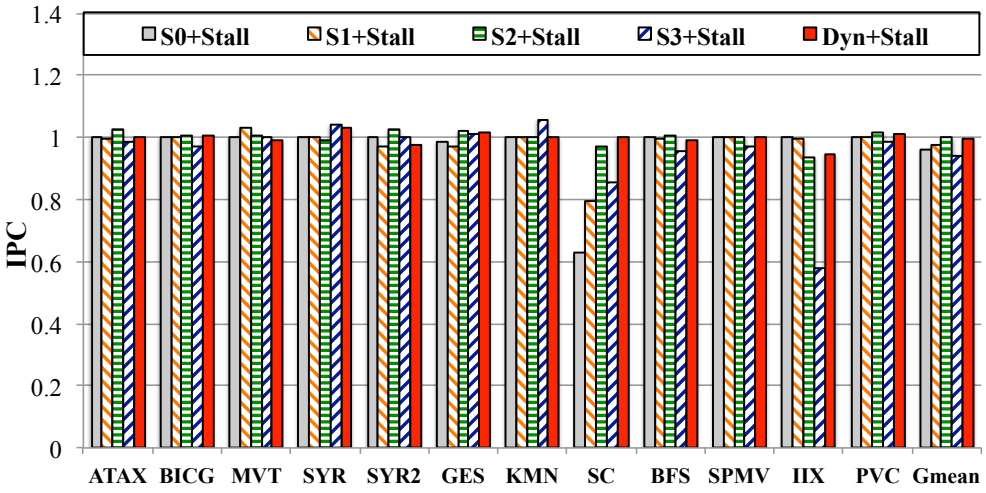


Figure 4.8: The impacts of using L1D Stalling to complement Constrained Replacement policy under static and dynamic partitioning. Results are normalized to corresponding partitioning schemes.

improvement of 25.9%, outperforming all static partitioning schemes among the evaluated benchmarks. We expect this dynamic partitioning scheme can adapt to other L1D configurations and future GPGPU benchmarks that have diverse branch and memory divergence.

4.3.5 Constrained Replacement with L1D Stalling

Figure 4.8 explains when stalling L1D can be an effective complement to replacement policy under static and dynamic partitioning. *SN* is equivalent to *StaticN* in Figure 4.7. The results are normalized to respective partitioning configurations. For example, *Dyn+Stall* is normalized to *Dyn* so that the impacts of stalling L1D can be explicitly presented. On average, enabling stall to complement replacement incurs 4.3%, 3.4%, 0%, and 6.9% performance loss in *S0+Stall*, *S1+Stall*, *S2+Stall*, and *S3+Stall*, respectively. Note that SC is the major contributor of the performance degradation. In SC, divergent loads are at the end of the kernel but out side of the main *for* loop body; stalling the divergent loads to prevent cache thrashing delays the completion of warps. On average, *Dyn+Stall* achieves 1% performance improvement. Since stalling L1D only works for a few partitioning schemes, we can conclude that L1D stalling is not suitable for GPU architecture to prevent cache thrashing.

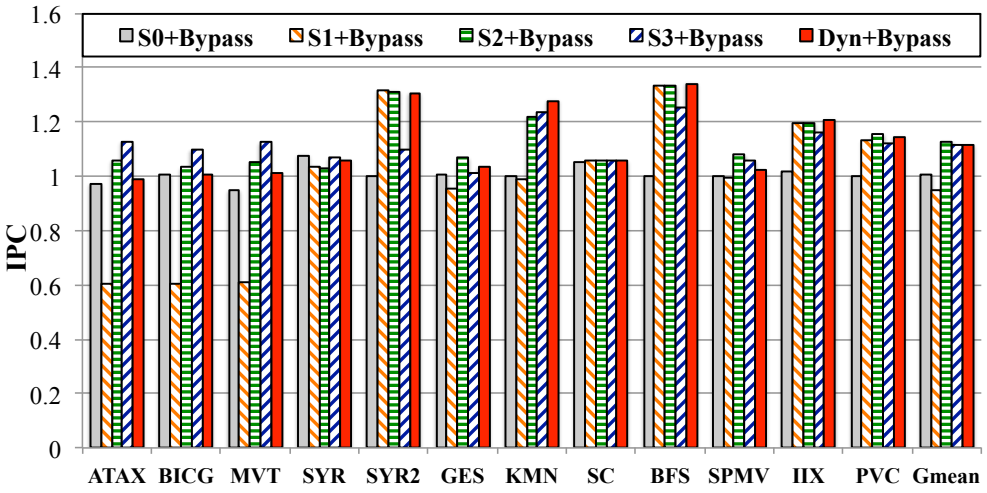


Figure 4.9: The impacts of using L1D Bypassing to complement Constrained Replacement policy under static and dynamic partitioning. Results are normalized to corresponding partitioning schemes.

4.3.6 Constrained Replacement with L1D Bypassing

Figure 4.9 explains when bypassing L1D can be an effective complement to replacement policy under static and dynamic partitioning. *SN* is equivalent to *StaticN* in Figure 4.7. The results are normalized to respective partitioning configurations. On average, constrained replacement with L1D bypassing incurs 0.6%, -5%, 12.8%, 11.4% and 11.6% performance improvement in *S0+Bypass*, *S1+Bypass*, *S2+Bypass*, *S3+Bypass*, and *Dyn+Bypass*, respectively. Note that these numbers are relative to partition-only configuration and are mainly used to quantify whether bypassing L1D is a viable complement to replacement policy. The performance degradation of *S1+Bypass* are mainly caused by ATAX, BICG, and MVT. We observe that the three benchmarks have a large amount of dead blocks in L1D. Aggressive bypassing slows down the removal of dead blocks so that cache capacity is underutilized.

4.3.7 Sensitivity to Promotion Granularity

Figure 4.10 analyzes the sensitivity of DaCache to the granularity in promotion policy. In this experiment, *Promo-MRU* immediately promotes re-referenced blocks to the MRU positions, while

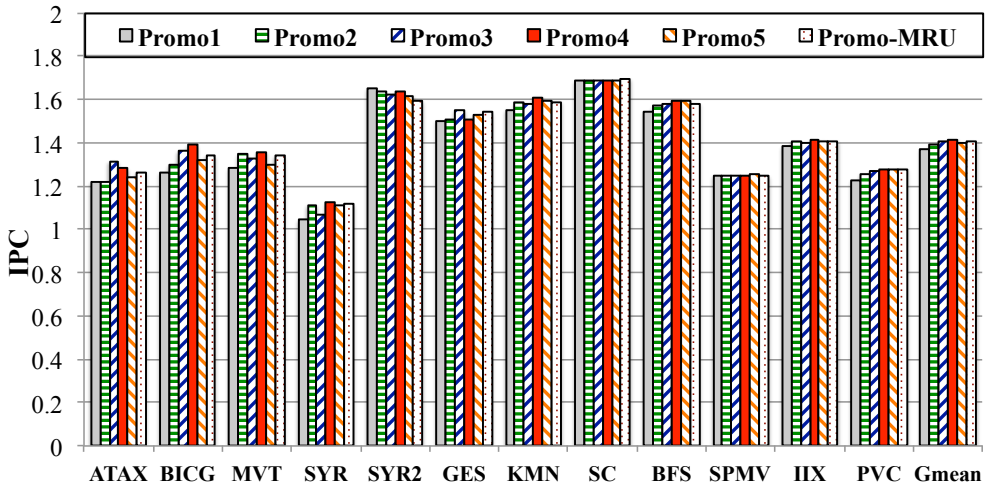


Figure 4.10: The impacts of promotion granularity under dynamic partitioning. *PromoN* means re-referenced blocks are promoted by *N* positions along the LRU-chain.

Promo1, *Promo2*, *Promo3*, *Promo4*, and *Promo5* promote re-referenced blocks by 1, 2, 3, 4, and 5 positions respectively along the LRU-chain unless they reach the MRU position. As we can see, the majority of the benchmarks are sensitive to promotion granularity. These dead blocks are gradually demoted into thrashing region by inserting new blocks and/or promoting re-referenced blocks into locality region. Thus, promotion granularity plays a critical role in eliminating dead blocks. Compared with LRU caches that directly promote re-referenced block to the MRU position, incremental promotion slowly promotes "hot" blocks towards the MRU position. The performance gap between *Promo1* (37.1%) and *Promo4* (40.4%) shows the importance of a fine-grained promotion policy in DaCache.

4.4 Related Work

There has been a large body of proposals on cache partitioning [29, 36, 37, 99, 11] and replacement policies [30, 65, 80] to increase the cache performance in CPU systems. However, these proposals do not handle the memory divergence issue within the massive parallelism of GPUs. Thus, we mainly review the latest work within the context of GPU cache management.

4.4.1 Cache Management for GPU Architecture

L1D bypassing has been adopted by multiple proposals to improve the efficiency of GPU caches. Jia *et al.* [40] observed that certain GPGPU access patterns experience significant intra-warp conflict misses due to the pathological behaviors in conventional cache indexing methods, and thus proposed a hardware structure called Memory Request Prioritization Buffer (MRPB). MRPB reactively bypasses L1D accesses that are stalled by cache associativity conflicts. Chen *et al.* [16] used extensions in L2 cache tag to track locality loss in L1D. If a block is requested twice by the same SM, it's assumed that severe contention happens in L1D so that replacement is temporarily locked down and new requests are bypassed into L2. Chen *et al.* proposed another adaptive cache management policy, Coordinated Bypassing and Warp Throttling (CBWT) [15]. CBWT uses protection distance prediction [18] to dynamically assign each new block a protection distance (PD), which guarantees that the block will not be evicted if its PD has not reached zero. When no unprotected lines are available, bypassing is triggered and the PD values are decreased. CBWT further throttles concurrency to prevent NOC from being congested by aggressive bypassing. Different from the above three techniques, bypassing L1D in DaCache is coordinated with warp scheduling logic and a finer-grained scheme to alleviate both inter- and intra-warp contention. At runtime, bypassing is limited to the thrashing region which caches divergent loads from warps with low scheduling priorities and coherent loads with no locality.

Compiler directed bypassing techniques have been investigated to improve GPU cache performance in [39, 98], but the static bypassing decisions mainly work for regular workloads. DaCache is a hardware solution for GPU cache management and can adapt to program behavior changes at runtime. In some heterogeneous multicore processors, CPU and GPU cores share the Last Level Cache (LLC). There are also some work on cache management for this kind of heterogeneous systems [49, 55]. Although DaCache is designed for discrete GPGPUs, the idea of coordinating warp scheduling and cache management is also applicable to hybrid CPU-GPU systems.

Dong proposed an AgeLRU algorithm [52] for GPU cache management. AgeLRU uses extra fields in cache tags to track each cache line's predicted reuse distance, reuse count, and the active

warp ID of the warp fetching the block, which together are used to calculate a score for replacement. The calculated score is reciprocal to each warp's age, i.e., older warps have higher scores to be protected. At runtime, the block with the lowest score is selected as replacement candidate and bypassing can be enabled when the score of the replacement victim is above a given threshold. By doing so, AgeLRU achieves the goal of preventing young warps from evicting blocks of old warps. DaCache doesn't need either storage in tag array or complicated calculation to assist replacement. By renovating the management policies, DaCache is more complexity-effective than AgeLRU to realize the same goal.

4.4.2 Warp Scheduling

There are several works that use warp scheduling algorithms to enable thrashing resistance in GPU data caches. Motivated by the observation that massive multithreading can increase contention in L1D for some highly cache-sensitive GPGPU benchmarks, Rogers et al. proposed a Cache Conscious Warp Scheduler (CCWS) [69] to limit the number of warps that issue load instructions when it detects loss of intra-warp locality. Following that, Rogers et al. also proposed a Divergence-Aware Warp Scheduling (DAWS) [70] to limit the number of actively scheduled warps whose aggregate cache footprint does not exceed L1D capacity. Kayiran et al. [44] proposed DYNCTA, a dynamic Cooperative Thread Array (CTA) scheduling mechanism. DYNCTA throttles the number of CTAs on each core according to application characteristics. Typically, DYNCTA reduces CTAs for data-intensive applications to minimizing resource contention. By throttling concurrency, cache contention can be alleviated, and Rogers et al. reported in [69] that warp scheduling can be more effective than optimal cache replacement [6] in preserving L1D locality. However, throttling concurrency usually permits only a few warps to be active, though each warp scheduler is hosting a lot more warps that are ready for execution (maximally 24 warps in our baseline). Our work is orthogonal to these warp scheduling algorithms, because contention still exists in reduced concurrency. DaCache can be used to increase cache utilization under reduced concurrency and also uplift the resultant concurrency.

4.5 Summary

GPUs are throughput-oriented processors that depend on massive multithreading to tolerate long latency memory accesses. The latest GPUs all are equipped with on-chip data caches to reduce the latency of memory accesses and save the bandwidth of NOC and off-chip memory modules. These tiny data caches are vulnerable to thrashing from massive multithreading, especially when divergent load instructions generate long bursts of cache accesses. Meanwhile, the blocks of divergent loads exhibit high intra-warp locality and are expected to be atomically cached so that the issuing warp can fully hit in L1D in the next load issuance. However, GPU caches are not designed with enough awareness of either lock-step execution model or memory divergence.

In this work, we renovate the cache management policies to design a GPU-specific data cache, *DaCache*. This design starts with the observation that warp scheduling can essentially shape the locality pattern in cache access streams. Thus, we incorporate the warp scheduling logic into insertion policy so that blocks are inserted into the LRU-chain according to their issuing warp's scheduling priority. Then, we deliberately prioritize coherent loads over divergent loads. In order to enable thrashing resistance, the cache ways are partitioned by desired warp concurrency into two regions, the locality region and the thrashing region, so that replacement is constrained within the thrashing region. When no replacement candidate is available in the thrashing region, incoming requests are bypassed. We also implement a dynamic partition scheme based on the caching effectiveness that is sampled at runtime. Experiments show that DaCache achieves 40.4% performance improvement over the baseline GPU and outperform two state-of-the-art thrashing resistant cache management techniques RRIP and DIP by 40% and 24.9%, respectively.

Chapter 5

OAWS: Memory Occlusion Aware Warp Scheduling

5.1 Introduction

To sustain the execution of a massive number of threads, GPUs are designed with GDDR5-based global memory that supports very high memory bandwidth. In addition, GPUs rely heavily on global memory coalescing to aggregate memory accesses from a warp of threads for higher bandwidth and fast access. Furthermore, the GPU scheduler strives to hide the latency of memory access by alternating the execution of many warps and overlapping the memory access of some warps with the computation of other warps.

GPU applications usually have a wide range of memory access patterns, many of which are very irregular. Despite the high-bandwidth GPU global memory, irregular patterns can still stall the GPU memory and degrade the effectiveness of massive parallelism. Recently GPUs have employed a hierarchy of data caches to reduce memory latency and save the on-chip network and off-chip memory bandwidth when there is locality within the accesses. However, the cache is frequently thrashed by divergent memory accesses. To tackle this problem, GPU L1D bypassing has been well studied to alleviate cache contention [40, 16, 15, 93, 51, 52, 103, 45]. For example, MRPB [40] aggressively bypasses L1D whenever an associativity stall occurs, but the cache is still underutilized. Two recent works [69, 70] reported that throttling the number of concurrent warps reduces the accumulated working set so that the contention on cache capacity is alleviated and locality is preserved.

Although the aforementioned efforts improved the memory performance of GPU applications, they overlooked some hazardous situations that are faced by GPU memory instructions. In this chapter, we have closely examined GPU resource utilization when executing data-intensive benchmarks. Our detailed analysis of GPU global memory accesses reveals that divergent accesses can

lead to the occlusion of Load-Store units due to rapid depletion of MSHR (Memory Status History Register) entries. This memory occlusion, in turn, stalls the execution pipeline, reducing GPU parallelism and degrading the overall performance. Our analysis shows that memory occlusion can significantly delay both coherent and divergent GPU instructions, exaggerate the memory stalls, and deteriorate the overall utilization of GPU.

Based on our analysis, we propose memory Occlusion Aware Warp Scheduling (OAWS) to monitor the usage of MSHR entries, predict the MSHR requirement of memory instructions, and schedule the warps that can be satisfied by the available MSHR entries, thereby preventing memory occlusion and increasing the effective parallelism among GPU warps. OAWS is designed with both static and dynamic methods to predict the required MSHR entries from GPU warps. Static OAWS predicts the misses from all warps with a fixed miss rate while dynamic OAWS takes into account the varying access patterns of different warps to predict cache misses on a per-warp basis. Particularly, dynamic OAWS has seamlessly integrated the warp priority and a concurrency estimation model in its prediction to prevent memory occlusion while preserving cache locality.

We have leveraged a wide variety of benchmarks to evaluate the performance of OAWS and demonstrated that OAWS outperforms three state-of-the-art warp scheduling techniques, i.e., CCWS [69], SWL [69], and MASCAR [73]. Specifically, our experiments show that our static and dynamic versions of OAWS achieve 35.3% and 74% compared to baseline GTO warp scheduling.

To the best of our knowledge, this body of work is the first to investigate the memory occlusion issue on GPU and address it through warp scheduling algorithms.

The rest of this chapter is organized as follows. Section 5.2 presents a high-level description of the proposed memory occlusion aware warp scheduling algorithms and the qualification metric. The static and dynamic prediction methods of OAWS are detailed in Sections 5.3 and 5.4, respectively. The experimental methodology and results are presented in Section 5.5. Section 5.6 summarizes the related work, followed by Section 5.7 that summarizes the chapter.

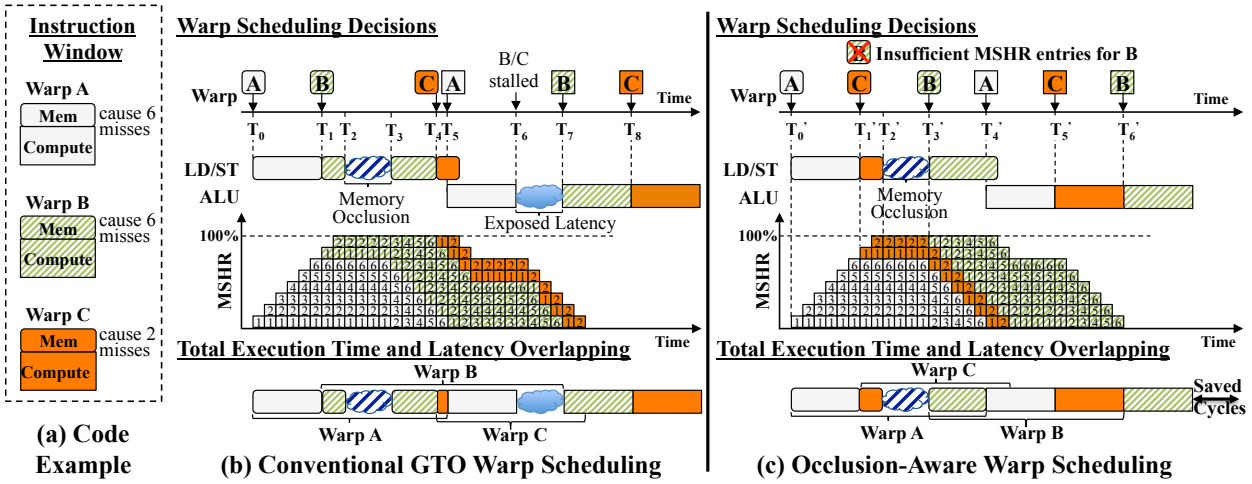


Figure 5.1: A conceptual example showing the benefits of Occlusion Aware Warp Scheduling

5.2 Main Idea of OAWS

As described in the previous section, divergent load instructions can deplete MSHR entries and occlude the LD/ST units. As a result, they significantly delay the execution of warps and exacerbate the long latency of off-chip memory operations. To address this problem, we propose memory Occlusion Aware Warp Scheduling (OAWS) that can monitor the execution of GPU warps, predict their MSHR requirement, and schedule the warps whose requirements can be met by the available MSHR entries. In doing so, OAWS can prevent memory occlusion, preserve L1D locality, and increase the effective warp-level parallelism.

Figure 5.1 provides an example on the benefit of OAWS. Assume that the example kernel first fetches data from memory and then computes it. There are 3 ready warps, A, B, and C, to be scheduled (Figure 5.1(a)) . They are ordered by their arrival times. The memory load instructions in warps A, B, and C are divergent and produce 6, 6, and 2 memory load requests to L2 cache, respectively. The varying number of memory requests for warps are caused by either branch divergence or warp locality variation [69, 70]. The LD/ST units have 8 MSHR entries. Unless otherwise noted, we employ by default the GTO [69] warp scheduler.

Figure 5.1(b) shows how the conventional GTO warp scheduler works. The memory load instructions from warps A, B, and C are issued at T_0, T_1, and T_4, respectively, according to their

age. By the time of T_2, two MSHR entries are allocated to two memory requests from warp B. At this time, MSHR entries are used up and memory occlusion occurs. Therefore, the remaining memory requests of warp B are continuously replayed until the responses for warp A's memory loads arrive at T_3. At T_5, all data required by warp A are returned from L2 and warp A is ready to be issued for computation. While warp A finishes computation at T_6, warps B and C are stalled due to outstanding memory requests. The computation for warps B and C starts at T_7 and T_8, respectively. As shown in the Figure 5.1(b), GTO causes idle cycles between T_6 and T_7.

In contrast, in Figure 5.1(c), OAWS predicts that available MSHR entries cannot satisfy warp B's requests, so it prevents warp B from issuing the memory instruction at $T_1^{'}$. Thus, OAWS avoids the memory occlusion caused by the depletion of MSHR entries by warp B. When warp A's requests return from memory, their MSHR entries are released and warp B is then scheduled. Since the ALU pipeline keeps idle, warp A and C are immediately scheduled for computation at $T_4^{'}$ and at $T_5^{'}$, respectively. Finally, warp B is scheduled for computation at $T_6^{'}$. Since warp C is scheduled ahead of warp B, its computation overlaps with warp B's memory load, reducing the time to complete these warps. Although this example shows only three warps and a miss penalty of 12 core cycles for simplicity, the benefit of stall-awareness could be more significant because the off-chip memory latency is often between 400 and 500 core cycles.

5.2.1 Qualification Metric of OAWS

OAWS aims to prevent memory occlusion with a simple logic, i.e., ensuring the available MSHR entries are more than the demand from a divergent load instruction. To this end, it needs to predict the number of cache misses for incoming divergent memory instructions. Meanwhile, because of the pipelined execution, several memory instructions could have been issued and will consume some MSHR entries. Thus, the qualification logic of OAWS is to qualify a memory instruction if its predicted cache misses can satisfy the following condition.

$$Miss_{pred}(pc, w) \leq Avail_{mshr} - Miss_{inflight} \tag{5.1}$$

where $Avail_{mshr}$ is the number of available MSHR entries, $Miss_{pred}(pc, w)$ is the predicted number of cache misses that warp w is going to incur for the memory instruction at address pc, and $Miss_{\text{inflight}}$ is the number of predicted cache misses from in-flight load instructions. At runtime, $Miss_{\text{inflight}}$ is updated as memory instructions are issued or completed by the LD/ST units. Since stores do not consume MSHR entries while coherent loads maximally consume two MSHR entries each time, they are always qualified for scheduling. Note that coherent load instructions rarely consume 2 entries. They are predicted to consume 1 MSHR entry with 1 cache miss. OAWS focuses on divergent load instructions which are more likely to cause memory occlusion.

5.2.2 Designing Scheduling Policies for OAWS

To accurately predict the number of cache misses for a divergent load instruction, we have explored both static and dynamic policies for OAWS. Accordingly, they are referred to as static OAWS and dynamic OAWS, respectively. Figure 5.2 presents the microarchitecture for both versions of OAWS. OAWS is implemented as an extension of the warp scheduler's qualification logic. Conventional qualification logic is used to pick warps that are ready for execution, which is denoted as a N-bit vector *Ready[1:N]* (❶). OAWS relies on the *Divergent Load Classifier (DLC)* to predict $Miss_{\text{pred}}(pc)$ and then re-qualifies ready warps using the logic in Equation 5.1. The output of OAWS is another N-bit vector *Occlude[1:N]* (❸), in which a bit value 0 denotes a warp predicted to not occlude MSHR entries, 1 otherwise. OAWS then uses the same prioritization logic as GTO to schedule the occlusion-free warps (❹). The following sections will introduce how static and dynamic OAWS are implemented.

5.3 Static OAWS

Static OAWS simply assigns a static miss rate (*SMR*) to each divergent load, then the predicted cache miss can be given as

$$Miss_{pred}(pc, w) = Div_{pred}(pc, w) \times SMR \tag{5.2}$$

where $Div_{pred}(pc, w)$ is the predicted memory divergence of a load instruction *pc* for warp *w* and *SMR* is the static L1D cache miss rate. This static method is proposed because memory instructions exhibit stable memory divergence behaviors in GPGPU workloads [70]. Div_{pred} for the divergent load instruction *pc* of warp *w* is equal to the number of active threads in *w*, similar to DAWS [70]. With *SMR* being 0%, OAWS is essentially disabled, and all divergent loads are assumed to complete without consuming MSHR entries. When *SMR* is 100%, OAWS assumes all divergent loads will cause cache misses in L1D, consuming MSHR entries. We tune *SMR* between 0% and 100% for the optimal performance.

In our experiments, we observe that static OAWS with a *SMR* of 50% achieves the optimal performance on the divergent benchmarks evaluated in this study. Given the SIMD width of 32 in our baseline GPU architecture, 50% means that each load is going to consume 16 MSHR entries. Since there are only 32 physical MSHR entries per SM, at most two divergent memory loads can be issued into the memory pipeline. Thus, the qualification logic in Equation 5.1 then serializes divergent memory instructions to access the memory pipeline. Because $Miss_{\text{inflight}}$ is decreased only when a load instruction retires from memory pipeline, this serialization conceptually inserts a minimum delay of $Div_{pred}(pc, w)$ cycles before a new divergent load can be qualified. When the miss rate is high, e.g., the remaining MSHR entries are less than 16, no divergent load can be scheduled and the delay is further extended. Such a serialization delay reduces the frequency of issuing divergent memory instructions, therefore static OAWS can alleviate the problem of memory occlusion.

To implement static OAWS, we only need to know whether a load instruction is divergent or not. This information is provided by *DLC*. In general, each *DLC* entry records the history of a divergent load, including the *PC* address, the number of instruction occurrences (*#inst*), the total number of memory accesses (*#acc*), and cache locality statistics (*#miss*). When a load instruction's memory accesses are coalesced (❺), its *PC* is first checked against *DLC* to make sure that no duplicate records exist in the table. If a new divergent load is detected, a new entry with current instruction's *PC*, *#inst* (being 1), and *#acc* are inserted into *DLC*; otherwise, the existing entry

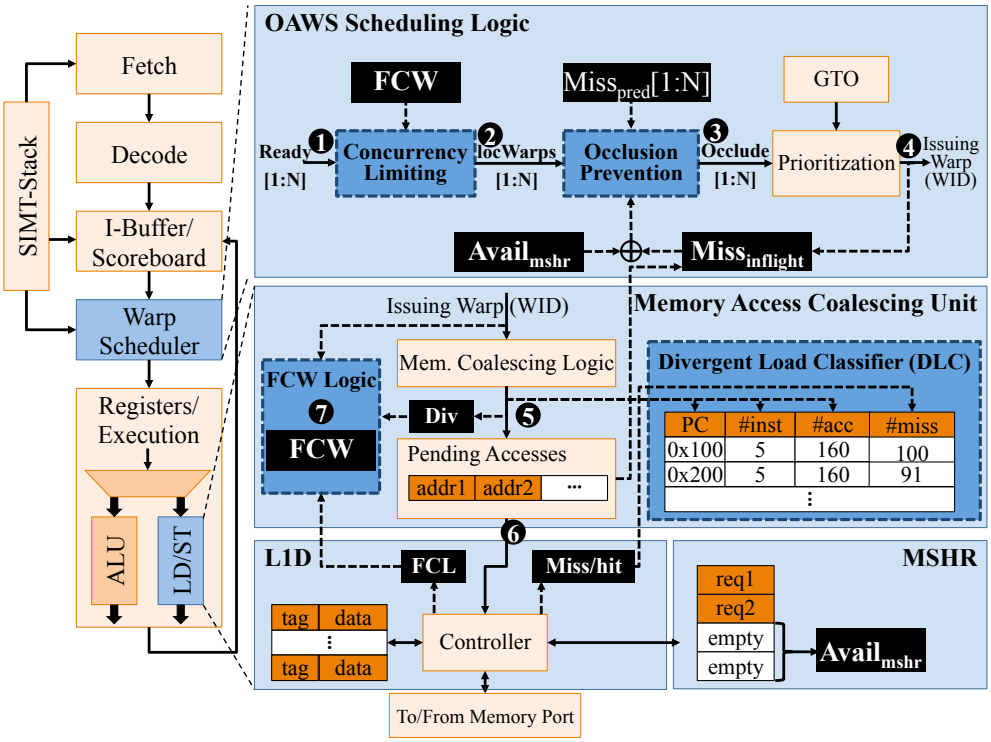


Figure 5.2: Detailed core model used for OAWS. N is the number of warp issue slots on the core.

is updated with this information. *#miss* comes from the cache hit/miss feedback (❻) when the individual divergent requests are processed by L1D. Since OAWS only focuses on divergent load instructions, coherent loads and their information are not populated into *DLC*. Static OAWS only needs the *PC* field to confirm if a load is divergent, while the other fields are used by dynamic OAWS.

5.4 Dynamic OAWS with L1D Locality Preservation

Static OAWS dictates a fixed miss rate for all warps, which cannot account for the dynamic nature of divergent accesses across different warps. In addition, the memory divergent benchmarks have high intra-warp locality. Thus, we propose dynamic OAWS to predict L1D cache misses on a per-warp basis, and then leverage these predicted misses to accomplish two objectives: (1) maximize the use of MSHR entries without memory occlusion and (2) maximize the number of concurrent warps while preserving L1D cache locality.

Both objectives require a careful selection of appropriate warps that can strike a balance between the maximum resource utilization (cache capacity and MSHR entries) and the best performance (no memory occlusion and no cache thrashing). We first propose a light-weight concurrency model to estimate the maximal number of warps that can make the LD/ST units occlusion-free. We employ then this concurrency model into dynamic OAWS.

5.4.1 Estimating Occlusion-Free Concurrency

The memory divergent benchmarks have high intra-warp locality so that they all can benefit from a throttled concurrency; meanwhile, warps that incur cache misses, regardless of the number of missed accesses, will inevitably thrash other warps' cache blocks and consume MSHR entries. Motivated by the above observation, we propose to use the number of fully cached loads to estimate such a concurrency that the aggregate MSHR consumption from all active warps is less than MSHR capacity. Being fully cached means a load's data is completely reserved in L1D and will incur neither cache eviction nor MSHR consumption. Under GTO prioritization, only a few old warps are actively scheduled, as shown in figure 2.7; thus, monitoring the dynamic changes of fully cached loads gives a close approximation of **Fully Cached Warps (FCW)**. We also observe that L1D locality in the data intensive benchmarks can be associated with both coherent and divergent loads. Since coherent loads often exhibit streaming-like accesses, we only use the locality statistics from divergent loads to estimate FCW.

Figure 5.3 shows the flow of FCW estimation based on the accumulated number of fully cached loads (*CNT*) and each warp's GTO scheduling priority (GTO_prio). At runtime, CNT is increased by 1 (Ⓐ) for each fully cached load. When CNT is saturated (CNT==Cmax), if FCW has not reached its maximum value (*Wmax*), FCW is increased by 1 and, accordingly, CNT is reset as *Cmax/2* to track fully cached divergent loads under the new concurrency (Ⓑ). For partially cached loads (Ⓒ), CNT is decreased differently depending on the issuing warp's scheduling priority. For instance, if a warp's scheduling priority is lower than FCW, CNT is decreased by 1 (Ⓓ); otherwise, CNT is decreased by FCW-GTO_prio (Ⓔ) to speed up the process of achieving the optimal FCW.

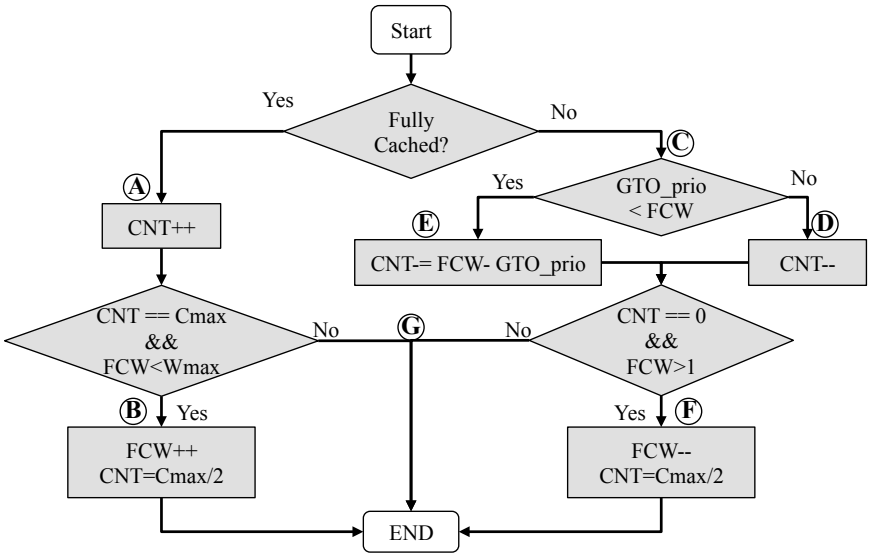


Figure 5.3: Flow Chart of FCW concurrency throttling logic.

When CNT reaches zero, FCW is decreased by 1 so that less warps are actively scheduled (Ⓕ). In our proposal, each warp scheduler enables at least one active warp (i.e., the oldest warp) to keep the on-chip resources busy, while *Wmax* is equal to 48, which is the number of physical warps on each SM. In the corner cases when FCW is 1 or Wmax (Ⓖ), CNT will not be overflowed if it is saturated. Such logic is a component inside MACU, denoted as *FCW Logic* in Figure 5.2.

In order to simplify its implementation, we first use one register (***Div*** in Figure 5.2) to mark whether a load is divergent or not, depending on the number of coalesced memory requests. *Div* is asserted when a new load instruction is serviced by L1D. We then use another register (***FCL*** in Figure 5.2) to track whether a load is fully cached or not. *FCL* is reset when L1D starts to service a new load, and is set when a cache miss happens on it. When all the accesses of the load are serviced, unset *FCL* indicates a fully cached load. This logic is triggered when a divergent load's accesses are serviced by L1D. We empirically use a 7-bit counter for CNT so that it can maximally record 128 consecutive occurrence of fully/partially cached loads, i.e., Cmax=128 in Figure 5.3. CNT is initialized as 64 while FCW is 1 for each warp scheduler. These initial values give FCW the flexibility to gradually learn L1D locality changes at runtime.

5.4.2 Concurrency-Aware Dynamic Prediction

We first use FCW to categorize all warps into two groups: **locality warps** and **thrashing warps**, based on their scheduling priorities. Assuming GTO priorities are in descending order, i.e., the oldest warp has the highest GTO priority of 0, a warp is a locality warp if its GTO priority is less than FCW, otherwise it is a thrashing warp. This classification is based on the observation that older warps are more frequently scheduled in GTO than younger ones so that they have higher L1D locality, and when a thrashing warp is scheduled, L1D locality is highly likely to be thrashed.

FCW can be used to throttle concurrency by only scheduling locality warps, which is referred to as *FCW Throttling*. However, the training process of FCW may take FCW Throttling a long time to arrive at the optimal value. This slow training process can be a disadvantage for FCW Throttling to quickly respond to program behavior changes during runtime. Memory occlusion may then occur. For example, when one locality warp fetches new data into L1D, it may deplete all available MSHR entries, occluding LD/ST units and preventing other locality warps from accessing L1D. Meanwhile, for benchmarks that have rich branch divergence, FCW Throttling may lose the opportunity to schedule thrashing warps with fewer active threads. Such kind of thrashing warps have small memory footprint and incur little cache thrashing. To prevent memory occlusion while maintaining sufficient concurrency, we integrate the use of FCW in dynamic OAWS for the prediction of cache misses, i.e., the required MSHR entries. The ultimate goal of dynamic OAWS is to disqualify some locality warps when memory occlusion occurs while qualifying some thrashing warps when the LD/ST units are occlusion-free.

In dynamic OAWS, we use the cache miss history of divergent memory instructions to predict the cache misses from a divergent load instruction. For locality warps, we predict their cache misses as

$$Miss_{pred}(pc,w) = Div_{pred}(pc,w) \times HMR(pc) \tag{5.3}$$

where $Div_{pred}(pc,w)$ is the predicted divergence as in Section 5.3 and $HMR(pc)$ is the history miss rate of current memory instruction at pc. In DLC, $HMR(pc)$ could be calculated as $\#misses/\#acc$.

For thrashing warps, we penalize them with a flat miss rate of 100%, i.e., their predicted cache misses is calculated as $Miss_{pred}(pc,w) = Div_{pred}(pc,w)$. This penalization means that the predicted number of misses is directly taken from the predicted memory divergence of thrashing warps. Setting such strict qualification standard for thrashing warps causes them to be scheduled when locality is high and outstanding memory requests are low.

By predicting cache misses based on *HMR*, the dynamic OAWS leverages the MSHR usage statistics from LD/ST units at its qualification logic to schedule warps. When the L1D cache is under-utilized due to a conservative FCW, more data accesses hit in the cache, therefore *HMR* is low. When *HMR* is low, more warps are predicted with fewer misses; therefore, more warps are scheduled for higher concurrency. Conversely, when the cache is over-subscribed with too many warps (FCW being too big), *HMR* is then very high, eventually bringing down the number of actively scheduled warps. This feedback-driven method eliminates the need of deducing program behavior changes or potential associativity conflicts. This feature helps OAWS achieve stable performance improvement across a wide range of workloads.

Per-instruction miss history gives uniform predictions for all active warps, which contradicts the observation that GTO prioritization creates skewed cache misses. In order to enforce the GTO scheduling prioritization as well as provide differentiated predictions for all active warps, we add each warp's GTO priority (a.k.a, warp ages) in addition to its history-based miss prediction. Without the need of adding extra logic into warp scheduler to enforce the threadblock-awareness, such as the proposal in OWL [42], we realize this goal through miss predictions. We amend our Equation 5.3 by adding GTO_{gprio} to the predicted misses, as shown below:

$$Miss_{pred}(pc,w) = Div_{pred}(pc,w) \times HMR(pc) + GTO_{gprio} \tag{5.4}$$

where GTO_{gprio} is a warp's global GTO priority within an SM. GTO_{gprio} is unique; thus, it could help provide differentiated predictions for all active warps among schedulers that share the same

set of MSHR entries. For locality warps, this linear prediction method matches the observation in Figure 2.7 that the cache misses of divergent loads are linear to warp ages under GTO scheduling.

Taken together, our dynamic OAWS strives to maintain a maximal number of concurrent warps and predict the demands on MSHR entries to prevent memory occlusion. These two objectives are seamlessly integrated into the dynamic prediction policy that takes into account the GTO priority, the history of per-instruction miss rate (*HMR*), and the estimated concurrency FCW.

5.4.3 Implementation and Overhead

We summarize other implementation details that have not been covered previously. Note that qualification logic is executed at a per-cycle basis. Calculating *HMR* at the same frequency based on the miss/hit statistics from DLC is impractical. Thus, we store each instruction's predicted cache misses into the instruction buffer, the same way as the ready bit for baseline qualification logic. By doing so, the process of miss prediction can be executed off the critical path of warp scheduling.

Divergent Load Classifier. The benchmarks we evaluate in this work typically have only one or two divergent loads in each kernel. But IIX from MapReduce [32] has 26 divergent loads in one of its kernels. Thus, we have 32 entries for DLC in both static and dynamic OAWS implementations. Because the fields of DLC are periodically updated, the entries that reach 0 (#inst) could be evicted. DLC could also be managed under a LRU policy for more complicated workloads. In addition, DLC is cleared at each kernel invocation.

Overhead Analysis. In order to implement the qualification logic in Equation 5.1, OAWS stores the information of both $Avail_{mshr}$ and $Miss_{inflight}$ into two registers. Our baseline GPU has 32 MSHR entries, and both registers need only 5 bits. FCW training needs two registers: 1-bit *Div* and 7-bit *FCL*. Our baseline GPU has a SIMD-width of 32. We use 5 bits to store the predicted cache misses for each instruction. Considering that there are 48 warps per SM and each warp can have two instructions, storing the predicted per-instruction cache misses has a total overhead of 480 bits (60 bytes). Each DLC entry needs 9 bytes, i.e., 40 bits for *PC*, 5 bits for *#inst*, 10 bits for

Table 5.1: Baseline GPGPU-Sim Configuration for OAWS Study

# of SMs	30 (15 clusters of 2)
SM Configuration	1400MHz, Reg #: 32K, Shared Memory: 48KB, SIMD Width: 16, warp: 32 threads, max threads per SM: 1024
Caches / SM	Data: 32KB/128B-line/8-way, Constant: 8KB/64B-line/24-way, Texture: 12KB/128B-line/2-way
Branching Handling	PDOM based method [21]
Warp Scheduling	GTO
Interconnect	Butterfly, 1400MHz, 32B channel width
L2 Unified Cache	768KB, 128B line, 16-way
Min. L2 Latency	120 cycles (compute core clock)
Cache Indexing	Pseudo-Random Hashing Function [58]
# Memory Partitions	6
# Memory Banks	16 per memory partition
Memory Controller	Out-of-Order (FR-FCFS), max request queue length: 32
Min. DRAM Latency	100 cycles (compute core clock)
GDDR5 Timing	$t_{CL} = 12$, $t_{RP} = 12$, $t_{RC} = 40$, $t_{RAS} = 28$, $t_{RCD} = 12$, $t_{RRD} = 6$, $t_{CDLR} = 5$, $t_{WR} = 12$

#acc, 10 bits for *#miss*, and 5 bits for *Div*. Thus, DLC table needs 288 bytes. In total, implementing OAWS needs 348-byte on-chip storage and four registers in each SM.

5.5 Experimental Evaluation

In this section, we will evaluate OAWS and discuss its design parameters for MSHR predication and warp scheduling.

5.5.1 Experimental Methodology

We use GPGPU-Sim [4] (version 3.2.1), a cycle-accurate simulator, to evaluate our OAWS mechanisms. The baseline GPU architectural parameters are summarized in Table 5.1. We use highly memory-divergent benchmarks from Rodinia [12], SHOC [17], PolyBench/GPU [28], Parboil [78], and MapReduce [32] for performance evaluation. These benchmarks' input sizes are listed in Table 5.2. We also evaluate the performance of OAWS on memory-coherent benchmarks from PolyBench/GPU [28], Rodinia [12], and Parboil [78]. All of the benchmarks are simulated

Table 5.2: Data-Intensive GPGPU (CUDA) Benchmarks for OAWS Study

#	Abbr.	Application	Suite	Input
Memory Divergent Benchmarks				
1	ATAX	matrix-transpose and vector mul.	PolyBench	$8K \times 8K$
2	BICG	kernel of BiCGStab linear solver	PolyBench	$8K \times 8K$
3	MVT	Matrix-vector-product transpose	PolyBench	8K
4	GES	Scalar-vector-matrix mul.	PolyBench	4K
5	SYRK	Symmetric rank-K operations	PolyBench	512×512
6	SYR2K	Symmetric rank-2K operations	PolyBench	256×256
7	KMN	Kmeans Clustering	Rodinia	28k 4x features
8	BFS	Breadth-First-Search	Rodinia	5M edges
9	SC	Streaming Cluster	Rodinia	256K Points
10	PF	Particle Filter	Rodinia	default
11	SPMV	Sparse matrix mul.	SHOC	default
12	IIX	Inverted Index	MapReduce	6.8M
Memory Coherent Benchmarks				
13	3DC	3D Convolution	PolyBench	default
14	2MM	2 Matrix Multiply	PolyBench	default
15	3MM	3 Matrix Multiply	PolyBench	default
16	FD	2D Finite Difference Kernel	PolyBench	default
17	BP	backprop	Rodinia	default
18	SRAD1	SRAD version 1	Rodinia	default
19	SRAD1	SRAD version 1	Rodinia	default
20	LBM	Lattrice-Boltzmann method	Parboil	default

Table 5.3: Configurations for SWL-Best and CCWS

SWL-Best				CCWS	
Bench.	Warps Actively Scheduled	Bench.	Warps Actively Scheduled	Name	Value
ATAX	2	SYR2K	2	$K_{THROTTLE}$	8
BICG	2	KMN	4	Victim Tag Array	8-way
MVT	2	BFS	3		16 entries per warp
GES	1	SPMV	2		(768 total entries)
SYRK	2	IIX	4	Warp Base Score	100

to completion and execute between 70 million and 1.5 billion instructions. The following warp scheduling algorithms are evaluated:

GTO is the baseline warp scheduler. All performance results are normalized to GTO.

Static Warp Limiting (SWL) [69] statically limits the number of warps that can be actively scheduled and needs to be tuned on a per-benchmark basis. Table 5.3 presents the warp-limiting value with the best performance for some memory divergent benchmark (*SWL-Best*). Other benchmarks are scheduled with maximal concurrency.

Cache Conscious Wavefront Scheduling (CCWS) [69] relies on a dedicated victim cache and a 6-bit Warp ID field in the tag of cache block to detect intra-warp locality and other storage to track per-warp locality changes. The warp that has the largest locality loss is exclusively prioritized. Configuration parameters for CCWS are summarized in Table 5.3.

Memory Aware Scheduling and Cache Access Re-execution (MASCAR) [73] exclusively prioritizes memory instructions from one "owner" warp when the memory subsystem is saturated; otherwise, memory instructions of all warps are prioritized over any computation instruction. MASCAR uses a re-execution queue to replay L1D accesses that are stalled due to MSHR unavailability or network congestion. Saturation here means that MSHR has only 1 entry or the queue inside memory port has only 1 slot. The re-execution queue has 32 entries.

Static OAWS (*OAWS-Static*) is described in Section 5.3. The default value for *SMR* is 50%. We will present the sensitivity analysis of *SMR* in Section 5.5.6.

Dynamic OAWS (*OAWS-Dyn*) is described in Section 5.4. *OAWS-Dyn* consists of two components, light-weight concurrency model and dynamic MSHR prediction. We also present the performance of using the light-weight concurrency model to manage concurrency, which is referred to as ***FCW-Only***.

5.5.2 Instructions Per Cycle (IPC)

Figure 5.4 shows IPC comparisons for our warp scheduling algorithms and the three state-of-the-art warp scheduling algorithms. We have the following key observations. First, *OAWS-Static* consistently improves performance for all memory divergent benchmarks, and on average achieves 35.3% IPC gains compared to baseline GTO scheduling and outperforms *MASCAR* by

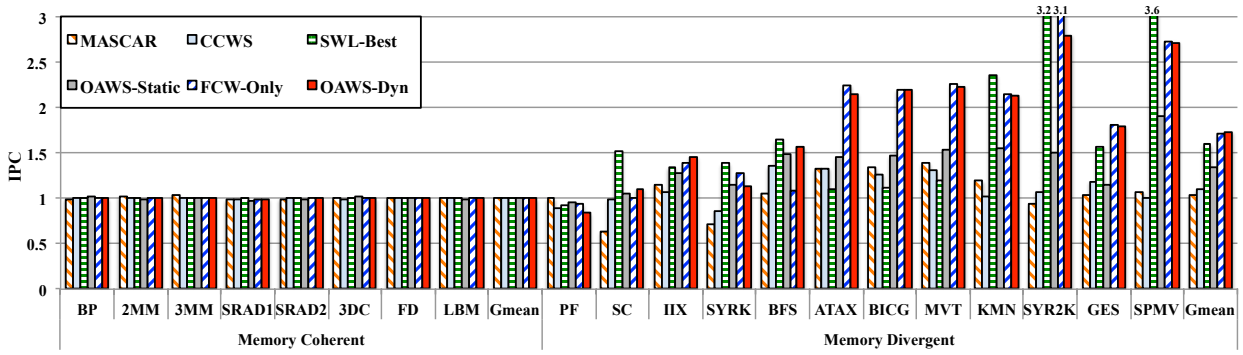


Figure 5.4: IPCs of various warp scheduling algorithms for memory coherent and memory divergent benchmarks. IPCs are normalized to the GTO scheduling.

29.2%. The performance improvement of *OAWS-Static* comes with the lowest hardware overhead, which strongly suggests the necessity and effectiveness of memory occlusion prevention. Second, by focusing on locality changes at the granularity of individual divergent loads, *FCW-Only* automatically increases or decreases concurrency to preserve effective concurrency, and therefore can outperform *CCWS* and *SWL-Best* by 56% and 7.7%, respectively. Finally, by conditionally loosening the concurrency limiting from *FCW-Only*, *OAWS-Dyn* significantly increases the performance of benchmarks that are branch divergent, such as IIX and BFS. Overall, *OAWS-Dyn* achieves 74% IPC improvement and outperforms *MASCAR*, *CCWS*, and *SWL-Best* by 65.8%, 57.2%, and 8.5%, respectively.

MASCAR and *CCWS* only improve performance by 4.8% and 10.4%, respectively, compared to the baseline. These low IPC gains can be explained from the following two aspects. First, we use an allocate-on-fill rather than an allocate-on-miss policy to manage L1D blocks on cache read misses. Given a 32-entry MSHR and 32KB L1D (256 blocks), the allocate-on-miss policy frequently reserves 32 cache blocks for outstanding memory requests in the memory divergent benchmarks, which wastes 12.5% of the L1D capacity. Second, reserving cache blocks can increase associativity conflicts in L1D. The memory divergent benchmarks from PolyBench/GPU [28] are highly sensitive to cache associativity [40]. Though we have applied the pseudo-random cache indexing function that is used in real Fermi architectures [58], associativity conflicts still have not been well mitigated. For example, each divergent load in ATAX generates 32 requests that are

constantly mapped into 8 out of the 32 sets in L1D. Consequently, allocate-on-miss policy exaggerates associativity conflicts. We have evaluated both policies for all experiments, and observed that *MASCAR* and *CCWS* perform similarly under the two policies, because they both provide exclusive accesses to one warp when the memory subsystem is saturated (*MASCAR*) or L1D locality loss is high (*CCWS*). However, GTO scheduling benefits greatly from increased effective L1D capacity and decreased associativity conflicts.

SWL-Best achieves 60% IPC improvement for all memory divergent benchmarks. The performance of *SWL-Best* could have been better if per-kernel tuning had been performed. For example, ATAX, BICG, and MVT have both coherent and divergent kernels. *SWL-Best*'s performance gains in divergent kernels are balanced out by the existence of coherent kernels where *SWL-Best* should not have been enabled. Meanwhile, *IIX* is a highly complicated benchmark with rich branch- and memory-divergence. IIX has 149 kernel invocations with the input size used in this evaluation. Given such complexity, *SWL-Best* has been included for pure performance comparison purposes.

For the coherent benchmarks, none of the warp scheduling algorithms achieve significant performance gains. Thus, we conclude that our proposals have no detrimental effects to memory-coherent benchmarks. In the following sections, we will dissect the performance of both *OAWS-Static* and *OAWS-Dyn* using memory divergent benchmarks only.

5.5.3 LD/ST Unit Stalls

Figure 5.5 breaks down LD/ST stall cycles when memory divergent benchmarks are scheduled by various scheduling algorithms. All numbers are normalized to LD/ST active cycles in GTO scheduling. Since all the evaluated benchmarks are read-intensive, memory occlusion caused by network congestion (*LDST_ICNT*) is negligible in every scheduling algorithm. Meanwhile, coalescing stalls (*LDST_COAL*) depend on memory divergence characteristics in the benchmarks. Warp scheduling can only impact stall cycles caused by *LDST_MSHR* and *MASCAR_Replay*. As we can see from Figure 5.5, *SWL-Best*, *OAWS-Static*, *FCW-Only*, and *OAWS-Dyn* have similar capability in reducing LD/ST stalls, which corresponds to their superior performance shown in

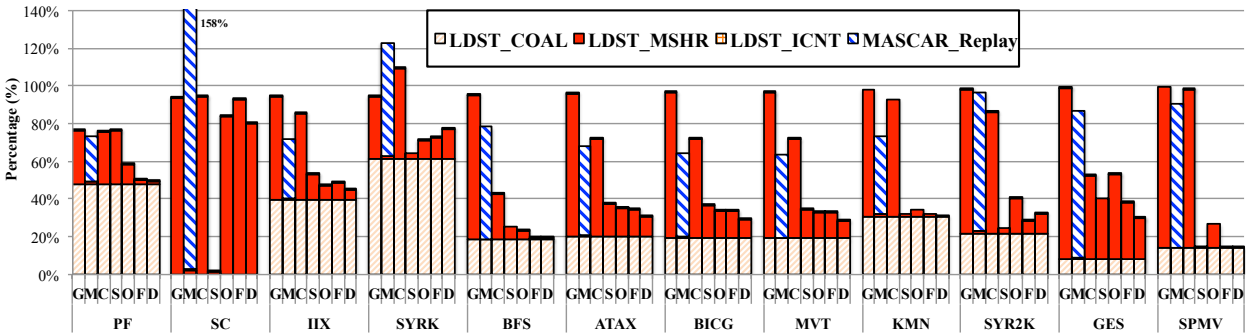


Figure 5.5: Breakdown of LD/ST stall cycles when the memory divergent benchmarks are scheduled by GTO (G), MASCAR (M), CCWS (C), SWL-Best (S), OAWS-Static (O), FCW-Only (F), and OAWS-Dyn (D). *MASCAR_Replay* only exists in MASCAR and refers to the cycles when the memory access from the re-execution queue can't be sent out.

Figure 5.4. Divergent loads in SC are references to arrays of structs and outside of a loop that generates high locality. Without careful concurrency throttling, divergent loads quickly thrash the locality of coherent loads. *FCW-Only* and *OAWS-Dyn* takes time to learn optimal concurrency, while *OAWS-Static* has no direct control over concurrency, therefore they perform equally poor in reducing LD/ST stalls for SC. When saturation in the memory subsystem is detected, MASCAR prevents memory accesses of "non-owner" warps from being sent out, i.e., replaying them until saturation is resolved. Such a strict requirement directly reduces LD/ST throughput, leading to the poor performance of MASCAR. *CCWS* prioritizes warps with the highest locality lost, which means that the prioritized warp often has little data reserved in L1D to start with and needs to immediately fetch data from L2. Switching prioritized warps incurs frequent MSHR consumptions, making CCWS less capable of LD/ST stall prevention.

5.5.4 Fully Cached Load Instructions

Within the lock-step execution, partially cached instructions still suffer from the problem of memory occlusion. Thus, we use the number of fully cached load instructions to quantify effective concurrency in the evaluated concurrency throttling mechanisms, i.e., *CCWS*, *SWL-Best*, *FCW-Only*, and *OAWS-Dyn*. Figure 5.6 presents the percentages of fully cached divergent loads. The results of GTO are also included as the baseline. *CCWS* has increased fully cached loads for all

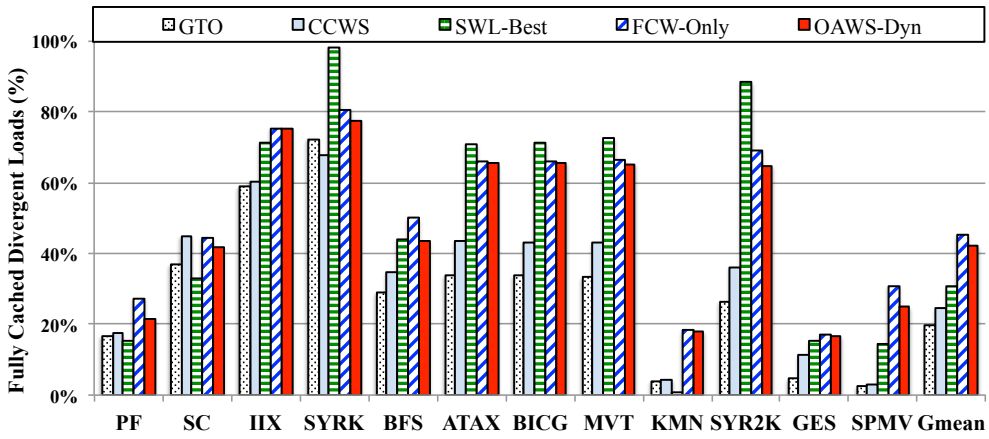


Figure 5.6: Percentage of fully cached divergent loads in the memory divergent benchmarks.

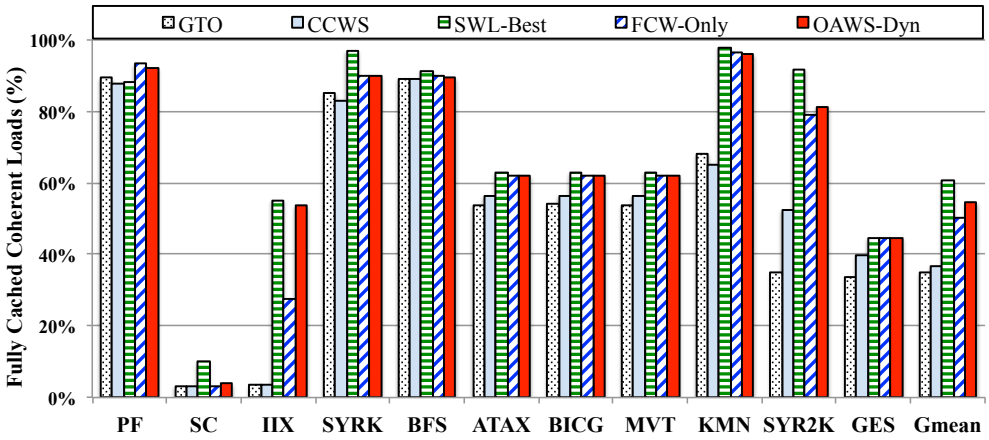


Figure 5.7: Percentage of fully cached coherent loads in the memory divergent benchmarks. SPMV has no coherent loads and is excluded from the figure.

benchmarks except SYRK. This is because SYRK has high inter-warp locality, while *CCWS* is specifically designed for intra-warp locality protection. The exclusively prioritized warp can make faster progress and evict its own data blocks that could have been utilized by other warps. *SWL-Best* achieves the best results for benchmarks with no branch divergence, such as SYRK, ATAX, BICG, MVT, and SYR2K. Due to high inter-warp locality, *SWL-Best* fully caches 98% and 88% of divergent loads in SYRK and SYR2K, respectively. For memory- and branch-divergent benchmarks, such as IIX and BFS, *FCW-Only* preserves more fully cached divergent loads than *SWL-Best*. Opportunistically scheduling more warps in *OAWS-Dyn* directly hampers locality preservation at instruction level, therefore it slightly underperforms compared to *FCW-Only* in the majority of the

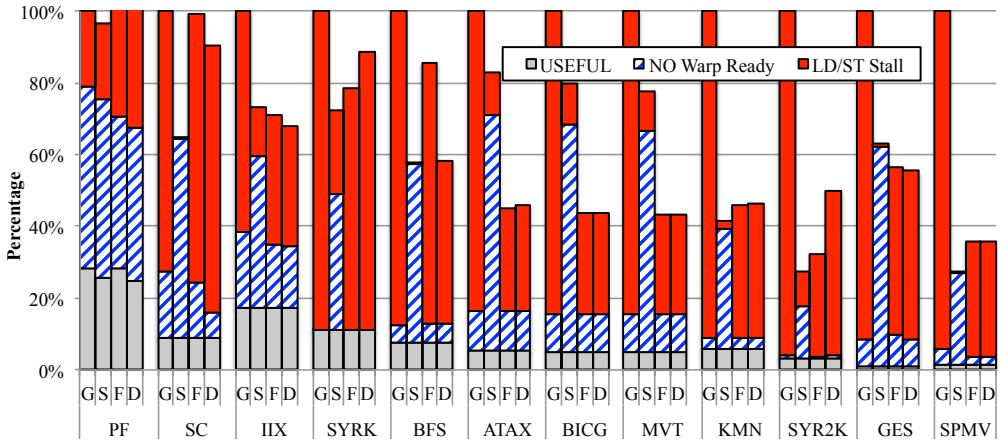


Figure 5.8: Breakdown of GPU cycles when memory divergent benchmarks are scheduled by GTO (G), SWL-Best (S), FCW-Only (F), and OAWS-Dyn (D).

benchmarks. On average, GTO, *CCWS*, *SWL-Best*, *FCW-Only*, and *OAWS-Dyn* fully cache 20%, 25%, 31%, 45%, and 42% of the total divergent load instructions, respectively.

Figure 5.7 presents the percentages of fully cached coherent loads. Some coherent loads in SC exhibit streaming accesses, making it have very low percentages of fully cached coherent loads under all scheduling algorithms. IIX has observable program behavior changes, i.e., coherent memory operations and large computation strictly follow divergent memory operations. Thus, *OAWS-Dyn* preserves more coherent loads in L1D than *FCW-Only*, which explains the performance advantage of *OAWS-Dyn* in IIX as shown in Figure 5.4. The other benchmarks are relatively simple, therefore the trend of fully cached coherent loads are similar to that in Figure 5.6.

The results in the two figures are highly correlated to the IPC discrepancy in Figure 5.4, except that *SWL-Best* does not have the best performance in ATAX, BICG, and MVT. This exception is caused by the fact that the three benchmarks have both memory divergent and memory coherent kernels as mentioned earlier.

5.5.5 Warp Scheduler Cycles

Figure 5.8 breaks down the GPU cycles when memory divergent benchmarks are scheduled by GTO, *SWL-Best*, *FCW-Only*, and *OAWS-Dyn*. This breakdown further reveals the different emphasis of the three concurrency throttling methods. Within the baseline GTO scheduling, schedulers

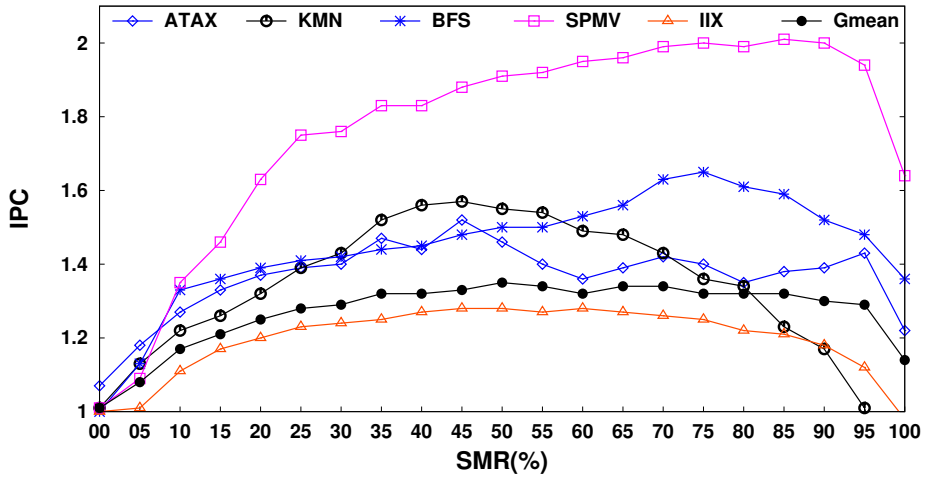


Figure 5.9: IPC of static OAWS with various *SMR* under five representative benchmarks.

are frequently stalled due to *LDST Stalls*. *SWL-Best* only enables a small pool of active warps to alleviate cache contention, which makes it suffer from an insufficient supply of fully cached warps. Thus, *SWL-Best* experiences a high percentage of *NO Warp Ready* cycles. When benchmarks have a mix of both coherent and divergent kernels, the static number of active warps is often suboptimal. The resultant concurrency is often excessive for divergent kernels but insufficient for coherent kernels. For example, when ATAX, BICG, and MVT are scheduled by *SWL-Best*, divergent kernels are limited by *LDST Stalls* cycles while coherent kernels are limited by *NO Warp Ready* cycles. Though *FCW-Only* and *OAWS-Dyn* have a larger pool of warps that are predicted to be free of LD/ST stalls, program behavior changes and the set-associative L1D together challenge their capability of finding optimal concurrency. The process of learning optimal concurrency in *FCW-Only* and *OAWS-Dyn* causes significantly more *LDST Stalls* cycles than the static and deterministic concurrency setting in *SWL-Best*. *OAWS-Dyn* has big reductions of *LDST Stalls* cycles for IIX and BFS, making it more feasible for real-world GPU workloads that have rich memory- and branch-divergence.

5.5.6 Sensitivity of Static OAWS to *SMR*

Figure 5.9 presents the IPC of static OAWS when *SMR* is swept from 0% to 100%, with an increment of 5%. When *SMR* is 0%, static OAWS is equal to GTO scheduling, which is the

baseline warp scheduling across all of the evaluations. Static OAWS achieves peak performance improvement (geometric mean: 35.3%), when *SMR* is 50%. In addition, in the case of *SMR* ranging from 35% to 90%, the static OAWS achieves stable IPC improvement with the standard deviation of 0.014, indicating its insensitivity to *SMR*. When *SMR* is larger than 50%, the qualification logic of static OAWS prevents two fully divergent load instructions being issued into memory stage, leading to such similar IPC improvements.

5.6 Related Work

Warp Scheduling plays a critical role in sustaining GPU performance and various scheduling algorithms have been proposed based on different heuristics. Among the concurrency throttling techniques, Static Warp Limiting (SWL) [69], Cache Conscious Warp Scheduling (CCWS) [69], and MASCAR [73] were discussed earlier in Section 5.5.1 and compared with our proposed scheduling mechanisms. On top of CCWS, Divergence Aware Warp Scheduling (DAWS) [70] actively schedules warps whose aggregate memory footprint does not exceed L1D capacity. The prediction of memory footprint requires compiler support to mark loops in the PTX ISA and other structures. Khairy *et al.* [45] proposed DWT-CS, which use core sampling to throttle concurrency. When L1D MPKI is above a given threshold, DWT-CS samples all SMs with different number of active warps and applies the best-performing active warp count on all SMs. Different from these concurrency throttling mechanisms, OAWS uses the number of fully cached divergent load instructions to dynamically adjust concurrency at a finer-granularity.

Some other warp scheduling algorithms are designed to improve GPU resource utilization. Fung *et al.* [21, 20] investigated the impact of warp scheduling on techniques aiming at branch divergence reduction, i.e., dynamic warp formation and threadblock compaction. Jog *et al.* [41] proposed an orchestrated warp scheduling to increase the timeliness of GPU L1D prefetching. Narasiman *et al.* [57] proposed a two-level round robin scheduler to prevent memory instructions from being issued consecutively. By doing so, memory latency can be better overlapped by computations. Gebhart *et al.* [23] introduced another two-level warp scheduler to manage a hierarchical

register file design. None of these warp scheduling techniques directly focuses on the problem of LD/ST stalls. On top of the two-level warp scheduling, Yu *et al.* [100] proposed a Stall-Aware Warp Scheduling (SAWS) to adjust the fetch group size when pipeline stalls are detected. SAWS mainly focuses on pipeline stalls, while OAWS is capable of avoiding LD/ST stalls and preserving L1D locality.

Kayiran et al. [44] proposed a dynamic Cooperative Thread Array (CTA) scheduling mechanism to enable the optimal number of CTAs according to application characteristics. It typically reduces concurrent CTAs for data-intensive applications to reduce LD/ST stalls. Lee et al. [50] proposed two alternative CTA scheduling schemes. Lazy CTA scheduling (LCS) utilizes a 3-phase mechanism to determine the optimal number of CTAs per core, while Block CTA scheduling (BCS) launches consecutive CTAs onto the same cores to exploit inter-CTA data locality. Jog et al. [42] proposed the OWL scheduler, which combines four component scheduling policies to improve L1D locality and the utilization of off-chip memory bandwidth. CTA scheduling is coarser than warp scheduling at concurrency throttling, therefore OAWS is better at locality preservation and LD/ST stall avoidance.

GPU Cache Management has been studied to preserve L1D locality, which can implicitly reduce LD/ST stalls. L1D bypassing is often adopted to alleviate cache contention. Jia *et al.* [40] designed a memory request buffer to reorder and prioritize L1D accesses and proposed to bypass L1D accesses that are stalled by cache associativity conflicts. Chen *et al.* [16] used extensions in L2 cache tag to track locality loss in L1D and bypass is temporarily triggered if a L2 cache block is requested twice by the same SM. Chen *et al.* [15] further proposed Coordinated Bypassing and Warp Throttling (CBWT) to orchestrate L1D bypassing and warp scheduling. Based on protection distance prediction [18], CBWT triggers bypassing when all L1D blocks are under protection and throttles concurrency to prevent NOC from being congested by aggressive bypassing. Wang *et al.* [93] proposed a DaCache design to orchestrate GPU cache management and warp scheduling. At runtime, DaCache bypasses divergent loads from warps with low scheduling priorities and coherent loads with no locality. Li *et al.* [51] proposed an locality monitoring mechanism to

bypass blocks that have low/no reuse or long reuse distances. Dong Li proposed an AgeLRU algorithm [52] to prevent young warps from evicting blocks of old warps. AgeLRU enables bypassing when the replacement score of the replacement candidate is above a given threshold. Based on CCWS, Dong Li [52] proposed another scheme named Priority-based Cache Allocation (PCAL) to bypass the memory accesses from non-prioritized warps so that other on-chip resources can be utilized. Based on DAWS, Zheng *et al.* [103] proposed Adaptive Cache and Concurrency (CCA) to bypass streaming memory accesses and accesses from inactive warps. Similarly, Khairy *et al.* [45] also proposed a technique to dynamically detect and bypass streaming memory accesses. Jia *et al.* and Xie *et al.* investigated compiler directed static bypassing techniques to improve GPU cache performance. Though these cache management schemes can ameliorate the problem of LD/ST stalls via preserved L1D locality, they are all reactive mechanisms. OAWS works from the source to prevent LD/ST stalls from occurring. Cache management schemes are orthogonal to OAWS and can be combined with OAWS to further improve GPU performance.

5.7 Summary

Efficient warp scheduling plays a critical role in sustaining high computation throughput of GPUs. In this chapter, we first identified a structural hazard caused by the lack of MSHR entries to meet the needs of divergent memory accesses in the GPU execution pipeline, which we referred to as memory occlusion. We then characterized and analyzed the impact of memory occlusion. To address the associated performance issues, we proposed a memory occlusion aware warp scheduling that could predict the demand of MSHR entries from GPU instructions and integrate this new knowledge in the qualification and prioritization logic of GPU warp schedulers to prevent memory occlusion. Both static and dynamic prediction methods have been designed and implemented to maximize the use of MSHR entries without memory occlusion while preserving L1D cache locality.

We have evaluated OAWS with static and dynamic prediction methods on a wide variety of memory divergent benchmarks. Static and dynamic OAWS techniques achieve 35.3% and

74% performance gains, respectively. Compared to state-of-the-art warp schedulers, i.e., MAS-CAR [73], CCWS [69], and SWL-Best [69], dynamic OAWS outperforms them by 65.8%, 57.2%, and 8.5%, respectively. In addition, OAWS is a pure hardware solution with minimal hardware cost, which makes it attractive in the cost-sensitive GPU chip industry. To our best knowledge, our work is the first to reveal the memory occlusion issue on GPU MSHR entries and propose occlusion aware warp scheduling algorithms to overcome its performance impact.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Leveraging the massive computation power of GPUs to accelerate data-intensive applications is a recent trend to embrace the arrival of the big data era. However, data-intensive applications often exhibit a wide variety of memory access patterns that cannot be coalesced to minimize memory traffic to high-bandwidth but long latency off-chip memory chips, posing great challenges on GPU resource management in hardware and GPU programmability in software. For example, memory divergence is a typical irregular memory access pattern that is well known to waste off-chip memory bandwidth. Due to the lack of architectural support to mitigate the impacts of memory divergence, accelerating data-intensive applications often needs tremendous programming efforts to optimize their memory accesses for high performance and throughput.

This dissertation has examined and quantified three new architectural bottlenecks that are closely associated with divergent memory operations, including intra-warp associativity conflicts, partial caching for high intra-warp cache locality, and memory occlusion. Meanwhile, this dissertation embodies a collection of research efforts to mitigate the performance impacts of these bottlenecks from their sources, including cache indexing method, data cache management policies, and warp scheduling logic. Based on the comprehensive experimental results and systematic comparisons with state-of-the-art techniques, this dissertation has made the following three key contributions.

1) Full-permutation Based GPU Cache Indexing: This dissertation has revealed that current cache indexing methods can pathologically cause severe intra-warp associativity conflicts for divergent memory accesses. It introduces a novel and simple Full-permutation (*FUP*) Based GPU

Cache Indexing method in Chapter 3 to disperse bursty intra-warp memory accesses into all available cache sets. It also introduces a new metric, intra-warp concentration, to quantify the distribution uniformity of intra-warp accesses into the cache sets. By minimizing intra-warp memory access concentration, FUP significantly eliminates intra-warp cache conflicts, improves L1D cache capacity utilization, and outperforms state-of-the-art cache indexing methods.

2) Memory Divergence-Aware GPU Cache Management: This dissertation has provided GPU L1 data cache management with memory divergence and warp scheduling awareness to efficiently reserve L1D locality and resist cache thrashing for memory-divergent applications. The proposed mechanism aims at two inefficiencies in current GPU cache management. First of all, divergent memory accesses often carry highly regular intra-warp locality, but they are often partially cached so that intra-warp locality is under-utilized to keep warp schedulers busy. Second, under GTO warp scheduling, old warps are more frequently scheduled and thus their data blocks have shorter re-reference intervals, but such valuable opportunity to approximate optimal cache replacement has been ignored. Accordingly, this dissertation has introduced Memory Divergence-Aware GPU Cache Management (*DaCache*) in the Chapter 4 to address these issues. DaCache mainly consists of two policies to resist both inter- and intra-warp thrashing. Based on an LRU cache, the Gauged Insertion policy provides multiple insertion positions in the LRU-chain for incoming data blocks. The insertion positions of blocks of divergent loads are determined by the issuing warp's scheduling priority, while blocks of coherent loads are inserted to MRU or LRU positions, depending on whether the coherent load instruction is detected to have locality or not. And the Constrained Replacement policy is used to enforce that only blocks with certain replacement priorities can be evicted. Our experimental evaluation with a diverse collection of workloads adequately demonstrates that DaCache substantially outperforms two state-of-the-art thrashing-resistant cache management techniques.

3) Memory Occlusion Aware Warp Scheduling: GPUs enable a high degree of Memory Level Parallelism (MLP) and Warp Level Parallelism (WLP) to overlap the long latency of off-chip memory operations. When memory divergence occurs, MSHR entries (representing MLP) can be

quickly depleted, and then divergent load instructions have to be replayed until all of the divergent accesses are processed by L1D. This dissertation names such scenarios as memory occlusion. The structural hazard due to the mismatch between limited MSHR capacity and bursty memory accesses incurs significant stall cycles in both LD/ST units and warp schedulers. This dissertation introduces a Memory Occlusion Aware Warp Scheduling (*OAWS*) to predict the MSHR consumption of divergent load instructions and then only schedule warps that will not incur memory occlusion. OAWS is implemented with both static and dynamic prediction methods for MSHR consumptions. Our experimental evaluation shows that OAWS substantially outperforms the state-of-the-art warp scheduling techniques.

By enabling architectural support for memory divergence mitigation, the techniques proposed in this dissertation can better prepare future GPU architectures to support more data-intensive applications with varied memory access patterns at ease.

6.2 Future Work

This dissertation has also opened up many opportunities for future architectural researches on optimizing data-intensive applications for efficient executions on GPUs. Particularly, the following two future studies are highly promising.

1) Compiler-Assisted GPU Cache Management: Our DaCache design (in Chapter 4) utilizes a small victim cache to detect whether a coherent load has intra-warp locality at runtime. This small structure incurs 172B storage overhead on each SM of our baseline GPU. Meanwhile, DaCache is based on the observation that divergent memory accesses are often associated with highly regular intra-warp L1D locality. However, the locality of divergent memory accesses may be unpredictable. Typically, the gather and scatter access pattern [33] often carries low or median L1D locality. For example, the BFS benchmark in Rodinia [12] has a divergent memory operation that has median intra- and inter-warp locality. This kind of load operations can thrash the locality potential of other memory operations, and make it hard to achieve an optimal partitioning for our constrained replacement policy.

The hardware structures that are used to detect such cache-unfriendly access patterns can be eliminated using compiler assistance. For NVIDIA GPUs with compute capability 2.0 or higher, CUDA PTX ISA (version 2.0) has introduced optional cache operators on load and store instructions to guide the management of L1 and L2 caches [62]. For example, the default cache operation for load instructions, *ld.ca*, caches data blocks in both L1 and L2 caches with normal eviction policy; *ld.cg* will bypass L1, while *ld.cs* triggers an evict-first policy for both L1 and L2 lines so that cache pollution from streaming/thrashing accesses can be reduced.

However, such cache operators can only enable coarse-grained caching decisions, i.e., "all-or-nothing" for a memory instruction in all warps. Under the massive parallelism of GPUs, it often takes non-trivial efforts to assign a proper cache operator for each memory instruction. More importantly, performance tuning often has portability issues across GPU devices. Our DaCache design can take the information from cache operators as hints to manage both L1 and L2 caches. Based on these hints, DaCache can further hybridize existing cache operators to manage cache locality at a finer-granularity. For example, for divergent load instructions, warps with high scheduling priorities can cache their data in L1, warps with median scheduling priorities can cache their data in L2, while warps with low scheduling priorities have to constantly fetch data from global memory. Store instructions can also be managed to reduce pollution at L2. This kind of scheduling-aware cache allocation scheme can essentially lead to a non-inclusive GPU cache hierarchy. Furthermore, the policies in DaCache can be further extended to manage sector-based GPU caches [68, 75, 72]. By doing so, cache thrashing due to massive parallelism can be further reduced and a higher number of active warps can be enabled to keep GPU resources busy.

2) Source-Throttling Congestion in On-chip Network: GPUs often employ two on-chip networks to transfer data between SM and L2 cache portions, one for data transfers from SM to L2 (downstream) and the other for data transfers from L2 to SM (upstream). Since GPGPU benchmarks are typically read-intensive, the upstream network that transfers requested data blocks is more vulnerable to be congested, especially when divergent memory accesses incur bursty memory traffic. Network congestion will not only prolong the latency of off-chip memory accesses,

but also prevent the L2/Memory Controller from accepting new requests. Consequently, the back-pressure originating from network congestion leads to severe under-utilization in various on-chip resources, such as the warp schedulers in each SM.

The request queues inside upstream network ports can be reorganized to enforce both warp-scheduling and memory-divergence awareness. Instead of injecting ready memory requests into upstream network in a FIFO manner, these requests can be assigned with different criticality and then serviced out-of-order. For example, coherent memory requests as well as divergent memory requests of high-priority warps can be prioritized. When network congestion occurs, only critical requests are injected into the upstream network. Based on this scheme, a more proactive method can be built to prevent network congestion from occurring.

The concepts of warp-scheduling awareness and memory-divergence awareness can also be applied to manage requests in the queues of L2 caches and memory controllers. Pending memory requests can be prioritized by issuing warps' scheduling priorities to access L2 caches and off-chip memory chips. In order to maximize the utilization of global memory bandwidth, this prioritization scheme needs to complement existing memory scheduling policies that aim at back-level parallelism and/or row-buffer locality.

Appendices

Appendix A

Publication Contributions

During my Ph.D. study, my research has contributed to the following publications (listed in the chronological order):

1. Xinyu Que, Weikuan Yu, Vinod Tipparaju, Jeffrey Vetter, and Bin Wang. Network-Friendly One-Sided Communication Through Multinode Cooperation on Petascale Cray XT5 Systems. In IEEE International Symposium on Cluster Computing and the Grid (CCGRID), 2011. [63]

2. Yuan Tian, Scott Klasky, Weikuan Yu, Hasan Abbasi, Bin Wang, Norbert Podhorszki, Ray W. Grout, and Matthew Wolf. A System-Aware Optimized Data Organization for Efficient Scientific Analytics. In The 21st International Symposium on High-Performance Parallel and Distributed Computing (HPDC), 2012. Poster Paper. [81]

3. Yuan Tian, Scott Klasky, Weikuan Yu, Hasan Abbasi, Bin Wang, Norbert Podhorszki, Ray W. Grout, and Matthew Wolf. SMART-IO: SysteM-AwaRe Two-Level Data Organization for Efficient Scientific Analytics. In International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2012. [82]

4. Zhuo Liu, Bin Wang, Patrick Carpenter, Dong Li, Jeffrey S. Vetter, and Weikuan Yu. PCM-Based Durable Write Cache for Fast Disk I/O. In International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2012. [54]

5. Yandong Wang, Yizheng Jiao, Cong Xu, Xiaobing Li, Teng Wang, Xinyu Que, Cristi Cira, Bin Wang, Zhuo Liu, Bliss Bailey, and Weikuan Yu. Assessing the Performance Impact of High-Speed Interconnects on MapReduce. In Third Workshop on Big Data Benchmarking (WBDB), 2012. [96]

6. Yuan Tian, Zhuo Liu, Scott Klasky, Bin Wang, Hasan Abbasi, Shujia Zhou, Norbert Podhorszki, Tom Clune, Jeremy Logan, and Weikuan Yu. A Lightweight I/O Scheme to Facilitate Spatial and Temporal Queries of Scientific Data Analytics. In IEEE Symposium on Massive Storage Systems and Technologies (MSST), 2013. [84]

7. Bin Wang and Weikuan Yu. Performance and Power Simulation for Versatile GPGPU Global Memory. In 27th IEEE International Parallel & Distributed Processing Symposium (IPDPS) PhD Forum, 2013. [92]

8. Yuan Tian, Scott Klasky, Weikuan Yu, Bin Wang, Hasan Abbasi, Norbert Podhorszki, and Ray Grout. DynaM: Dynamic Multiresolution Data Representation for Large-Scale Scientific Analysis. In IEEE International Conference on Networking, Architecture, and Storage (NAS), 2013. [83]

9. Bin Wang, Yizheng Jiao, Weikuan Yu, Xipeng Shen, Dong Li, and Jeffrey Vetter. A Versatile Performance and Energy Simulation Tool for Composite GPU Global Memory. In IEEE 21st International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2013. [89]

10. Bin Wang, Bo Wu, Dong Li, Xipeng Shen, Weikuan Yu, Yizheng Jiao, and Jeffrey Vetter. Exploring Hybrid Memory for GPU Energy Efficiency through Software-Hardware Co-Design. In The 22nd International Conference on Parallel Architecture and Compilation Techniques (PACT), 2013. [91]

11. Bin Wang, Zhuo Liu, Xinning Wang, and Weikuan Yu. Eliminating Intra-Warp Conflict Misses in GPU. In The 18th Design, Automation and Test in Europe (DATE), 2015. [90]

12. Bin Wang, Weikuan Yu, Xian-He Sun and Xinning Wang. DaCache: Memory Divergence-Aware GPU Cache Management. In The 29th International Conference on Supercomputing (ICS), 2015. [93]

13. Teng Wang, Sarp Oral, Michael Pritchard, Bin Wang, and Weikuan Yu. TRIO: Burst Buffer Based I/O Orchestration. In IEEE International Conference on Cluster Computing, 2015. [94]

14. Xinning Wang, Bin Wang, Zhuo Liu, and Weikuan Yu. Preserving Row Buffer Locality for PCM Wear-Leveling Under Massive Parallelism. In IEEE 23rd International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2015. [95]

15. Bin Wang, Weikuan Yu, and Jianhui Yue. OAWS: Memory Occlusion Aware Warp Scheduling. Under review.

Bibliography

[1] Tor M. Aamodt and Wilson W. L. Fung. GPGPUSim 3.x Manual. http://gpgpu-sim.org/manual/index.php/GPGPU-Sim_3.x_Manual, 2014.

[2] Anant Agarwal and Steven D. Pudar. Column-associative Caches: a Technique for Reducing the Miss Rate for Direct-Mapped Caches. In *Proceedings of the 20th International Symposium on Computer Architecture (ISCA)*, pages 179–190, 1993.

[3] Joshua A. Anderson, Chris D. Lorenz, and A. Travesset. General Purpose Molecular Dynamics Simulations Fully Implemented on Graphics Processing Units. *Journal of Computational Physics*, 227(10):5342–5359, May 2008.

[4] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *Proceedings of the 2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 163–174, 2009.

[5] Peter Bakkum and Kevin Skadron. Accelerating SQL Database Operations on a GPU with CUDA. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*, pages 94–103, 2010.

[6] Laszlo A. Belady. A Study of Replacement Algorithms for a Virtual Storage Computer. *IBM Systems Journal*, 5(2):78–101, June 1966.

[7] François Bodin and André Seznec. Skewed Associativity Improves Program Performance and Enhances Predictability. *IEEE Transactions on Computers*, 46(5):530–544, 1997.

[8] Brett W. Coon and Peter C. Mills and Stuart F. Oberman and Ming Y. Siu. Tracking register usage during multithreaded processing using a scoreboard having separate memory regions and storing sequential register size indicators, October 7 2008. US Patent 7,434,032.

[9] Nicolas Brunie, Sylvain Collange, and Gregory Frederick Diamos. Simultaneous Branch and Warp Interweaving for Sustained GPU Performance. In *Proceedings of the 39th International Symposium on Computer Architecture (ISCA)*, pages 49–60, 2012.

[10] Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. A Map Reduce Framework for Programming Graphics Processors. In *Proceedings of Workshop on Software Tools for MultiCore Systems*, 2008.

[11] Jichuan Chang and Gurindar S. Sohi. Cooperative Cache Partitioning for Chip Multiprocessors. In *Proceedings of the 21th Annual International Conference on Supercomputing (ICS)*, pages 242–252, 2007.

[12] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.

[13] Linchuan Chen and Gagan Agrawal. Optimizing MapReduce for GPUs with Effective Shared Memory Usage. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 199–210, 2012.

[14] Linchuan Chen, Xin Huo, and Gagan Agrawal. Accelerating MapReduce on a Coupled CPU-GPU Architecture. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 25:1–25:11, 2012.

[15] Xuhao Chen, Li-Wen Chang, Christopher I. Rodrigues, Jie Lv, Zhiying Wang, and Wen mei W. Hwu. Adaptive Cache Management for Energy-Efficient GPU Computing. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 343–355, 2014.

[16] Xuhao Chen, Shengzhao Wu, Li-Wen Chang, Wei-Sheng Huang, Carl Pearson, Zhiying Wang, and Wen mei W. Hwu. Adaptive Cache Bypass and Insertion for Many-core Accelerators. In *Proceedings of the 2nd International Workshop on Many-core Embedded Systems (MES)*, pages 1–8, 2014.

[17] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*, pages 63–74, 2010.

[18] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V. Veidenbaum. Improving Cache Management Policies Using Dynamic Reuse Distances. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 389–400, 2012.

[19] Jean Marc Frailong, William Jalby, and Jacques Lenfant. XOR-Schemes: A Flexible Data Organization in Parallel Memories. In *Proceedings of the 14th International Conference on Parallel Processing (ICPP)*, pages 276–283, 1985.

[20] Wilson W. L. Fung and Tor M. Aamodt. Thread Block Compaction for Efficient SIMT Control Flow. In *Proceedings of the 17th International Conference on High-Performance Computer Architecture (HPCA)*, pages 25–36, 2011.

[21] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 407–420, 2007.

[22] Hongliang Gao and Chris Wilkerson. A Dueling Segmented LRU Replacement Algorithm with Adaptive Bypassing. In *JWAC 2010 - 1st JILP Worshop on Computer Architecture Competitions: cache replacement Championship*, 2010.

[23] Mark Gebhart, Daniel R. Johnson, David Tarjan, Stephen W. Keckler, William J. Dally, Erik Lindholm, and Kevin Skadron. Energy-Efficient Mechanisms for Managing Thread Context in Throughput Processors. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 235–246, 2011.

[24] Tony Givargis. Improved Indexing for Cache Miss Reduction in Embedded Systems. In *Proceedings of the 40th Design Automation Conference (DAC)*, pages 875–880, 2003.

[25] Antonio González, Mateo Valero, Nigel Topham, and Joan M. Parcerisa. Eliminating Cache Conflict Misses Through XOR-based Placement Functions. In *Proceedings of the 11th International Conference on Supercomputing (ICS)*, pages 76–83, 1997.

[26] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. Gputerasort: High performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 325–336, 2006.

[27] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast Computation of Database Operations Using Graphics Processors. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 215–226, 2004.

[28] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a High-Level Language Targeted to GPU Codes. In *Proceedings of Innovative Parallel Computing (InPar)*, pages 1–10, 2012.

[29] Fei Guo, Yan Solihin, Li Zhao, and Ravishankar Iyer. A Framework for Providing Quality of Service in Chip Multi-Processors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 343–355, 2007.

[30] Erik G. Hallnor and Steven K. Reinhardt. A Fully Associative Software-Managed Cache Design. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*, pages 107–116, 2000.

[31] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue B. Moon. PacketShader: a GPU-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 195–206, 2010.

[32] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: a MapReduce Framework on Graphics Processors. In *Proceedings of the 17th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 260–269, 2008.

[33] Bingsheng He, Naga K. Govindaraju, Qiong Luo, and Burton Smith. Efficient Gather and Scatter Operations on Graphics Processors. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC)*, pages 1–12, 2007.

[34] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational Query Coprocessing on Graphics Processors. *ACM Transactions on Database Systems*, 34(4):21:1–21:39, December 2009.

[35] David T. Harper III and J. Robert Jump. Vector Access Performance in Parallel Memories Using A Skewed Storage Scheme. *IEEE Transactions on Computers*, C-36(12):1440–1449, December 1987.

[36] Ravi Iyer. CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms. In *Proceedings of the 18th Annual International Conference on Supercomputing (ICS)*, pages 257–266, 2004.

[37] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. QoS Policies and Architecture for Cache/Memory in CMP Platforms. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 25–36, 2007.

[38] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel S. Emer. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP). In *Proceedings of the 37th International Symposium on Computer Architecture (ISCA)*, pages 60–71, 2010.

[39] Wenhao Jia, Kelly A. Shaw, and Margaret Martonosi. Characterizing and Improving the Use of Demand-fetched Caches in GPUs. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS)*, pages 15–24, 2012.

[40] Wenhao Jia, Kelly A. Shaw, and Margaret Martonosi. MRPB: Memory Request Prioritization for Massively Parallel Processors. In *Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 272–283, 2014.

[41] Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. Orchestrated Scheduling and Prefetching for GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, pages 332–343, 2013.

[42] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance. In *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 395–406, 2013.

[43] Gary J. Katz and Joseph T. Kider Jr. All-Pairs Shortest-Paths for Large Graphs on the GPU. In *Proceedings of the EUROGRAPHICS/ACM SIGGRAPH Conference on Graphics Hardware*, pages 47–55, 2008.

[44] Onur Kayiran, Adwait Jog, Mahmut T. Kandemir, and Chita R. Das. Neither More nor Less: Optimizing Thread-level Parallelism for GPGPUs. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 157–166, 2013.

[45] Mahmoud Khairy, Mohamed Zahran, and Amr G. Wassal. Efficient Utilization of GPGPU Cache Hierarchy. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs (GPGPU)*, pages 36–47, 2015.

[46] Samira Manabi Khan, Yingying Tian, and Daniel A. Jimenez. Sampling Dead Block Prediction for Last-Level Caches. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 175–186, 2010.

[47] Mazen Kharbutli, Keith Irwin, Yan Solihin, and Jaejin Lee. Using Prime Numbers for Cache Indexing to Eliminate Conflict Misses. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture (HPCA)*, pages 288–299, 2004.

[48] Duncan H. Lawrie and Chandra R. Vora. The Prime Memory System for Array Access. *IEEE Transactions on Computers*, 31(5):435–442, 1982.

[49] Jaekyu Lee and Hyesoon Kim. TAP: A TLP-Aware Cache Management Policy for a CPU-GPU Heterogeneous Architecture. In *Proceedings of the 18th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 91–102, 2012.

[50] Minseok Lee, Seokwoo Song, Joosik Moon, John Kim, Woong Seo, Yeon-Gon Cho, and Soojung Ryu. Improving GPGPU Resource Utilization Through Alternative Thread Block Scheduling. In *Processing of the 20th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 260–271, 2014.

[51] Chao Li, Shuaiwen Song, Hongwen Dai, Albert Sidelnik, Siva Kumar Sastry Hari, and Huiyang Zhou. Locality-Driven Dynamic GPU Cache Bypassing. In *Proceedings of the 29th International Conference on Supercomputing (ICS)*, pages 128–139, 2015.

[52] Dong Li. *Orchestrating Thread Scheduling and Cache Management to Improve Memory System Throughput in Throughput Processor*. PhD thesis, University of Texas at Austin, May 2014.

[53] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, March 2008.

[54] Zhuo Liu, Bin Wang, Patrick Carpenter, Dong Li, Jeffrey S. Vetter, and Weikuan Yu. PCM-Based Durable Write Cache for Fast Disk I/O. In *Proceedings of the 20th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 451–458, 2012.

[55] Vineeth Mekkat, Anup Holey, Pen-Chung Yew, and Antonia Zhai. Managing Shared Last-Level Cache in a Heterogeneous Multicore Processor. In *Proceedings of the 22nd international conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 225–234, 2013.

[56] Peter C. Mills, John Erik Lindholm, Brett W. Coon, Gary M. Tarolli, and John Matthew Burgess. Scheduler in multi-threaded processor prioritizing instructions passing qualification rule, May 24 2011. US Patent 7,949,855.

[57] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. Improving GPU Performance via Large Warps and Two-level Warp Scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 308–317, 2011.

[58] Cedric Nugteren, Gert-Jan van den Braak, Henk Corporaal, and Henri Bal. A Detailed GPU Cache Model Based on Reuse Distance Theory. In *Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 37–48, 2014.

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

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

[61] NVIDIA. CUDA C Programming Guide, 2013.

[62] NVIDIA. PTX: Parallel Thread Execution ISA Version 4.2, 2015.

[63] Xinyu Que, Weikuan Yu, Vinod Tipparaju, Jeffrey S. Vetter, and Bin Wang. Network-Friendly One-Sided Communication through Multinode Cooperation on Petascale Cray XT5 Systems. In *Proceedings of 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 352–361, 2011.

[64] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, Jr., and Joel S. Emer. Adaptive Insertion Policies for High Performance Caching. In *Proceedings of the 34th International Symposium on Computer Architecture (ISCA)*, pages 381–391, 2007.

[65] Moinuddin K. Qureshi, David Thompson, and Yale N. Patt. The V-Way Cache: Demand Based Associativity via Global Replacement. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, pages 544–555, 2005.

[66] Ram Raghavan and John P. Hayes. On Randomly Interleaved Memories. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, pages 49–58, 1990.

[67] B. Ramakrishna Rau. Pseudo-Randomly Interleaved Memory. In *Proceedings of the 18th Annual International Symposium on Computer Architecture (ISCA)*, pages 74–83, 1991.

[68] Minsoo Rhu, Michael Sullivan, Jingwen Leng, and Mattan Erez. A Locality-Aware Memory Hierarchy for Energy-Efficient GPU Architectures. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 86–98, 2013.

[69] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. Cache-Conscious Wavefront Scheduling. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 72–83, 2012.

[70] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. Divergence-aware Warp Scheduling. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 99–110, 2013.

[71] Alberto Ros, Polychronis Xekalakis, Marcelo Cintra, Manuel E. Acacio, and José M. García. ASCIB: Adaptive Selection of Cache Indexing Bits for Removing Conflict Misses. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, pages 51–56, 2012.

[72] Jeffrey B. Rothman and Alan Jay Smith. The Pool of Subsectors Cache Design. In *Proceedings of the 13th International Conference on Supercomputing (ICS)*, pages 31–42, 1999.

[73] Ankit Sethia, Davoud Anoushe Jamshidi, and Scott A. Mahlke. Mascar: Speeding up GPU Warps by Reducing Memory Pitstops. In *Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 174–185, 2015.

[74] André Seznec. A Case for Two-Way Skewed-Associative Caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA)*, pages 169–178, 1993.

[75] André Seznec. Decoupled Sectored Caches: Conciliating Low Tag Implementation Cost. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA)*, pages 384–393, 1994.

[76] André Seznec. A new case for skewed associativity. Technical report, IRISA Technical Report 1114, 1997.

[77] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science and Engineering*, 12(3):66–73, May 2010.

[78] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Changx, Nasser Anssari, Geng Daniel Liu, and Wen mei W. Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. *IMPACT Technical Report, IMPACT-12-01, University of Illinois, at Urbana-Champaign*, 2012.

[79] Jeff A. Stuart and John D. Owens. Multi-GPU MapReduce on GPU Clusters. In *Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, pages 1068–1079, 2011.

[80] Ranjith Subramanian, Yannis Smaragdakis, and Gabriel H. Loh. Adaptive Caches: Effective Shaping of Cache Behavior to Workloads. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 385–396, 2006.

[81] Yuan Tian, Scott Klasky, Weikuan Yu, Hasan Abbasi, Bin Wang, Norbert Podhorszki, Ray W. Grout, and Matthew Wolf. A System-Aware Optimized Data Organization for Efficient Scientific Analytics. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 125–126, 2012.

[82] Yuan Tian, Scott Klasky, Weikuan Yu, Hasan Abbasi, Bin Wang, Norbert Podhorszki, Ray W. Grout, and Matthew Wolf. SMART-IO: SysteM-AwaRe Two-Level Data Organization for Efficient Scientific Analytics. In *Proceedings of IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 181–188, 2012.

[83] Yuan Tian, Scott Klasky, Weikuan Yu, Bin Wang, Hasan Abbasi, Norbert Podhorszki, and Ray W. Grout. DynaM: Dynamic Multiresolution Data Representation for Large-Scale Scientific Analysis. In *Proceedings of IEEE Eighth International Conference on Networking, Architecture and Storage (NAS)*, pages 115–124, 2013.

[84] Yuan Tian, Zhuo Liu, Scott Klasky, Bin Wang, Hasan Abbasi, Shujia Zhou, Norbert Podhorszki, Tom Clune, Jeremy Logan, and Weikuan Yu. A Lightweight I/O Scheme to Facilitate Spatial and Temporal Queries of Scientific Data Analytics. In *Proceedings of IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2013.

[85] top500.org. TOP 10 Sites for June 2015. http://www.top500.org/lists/2015/06, 2015.

[86] Nigel P. Topham, Antonio González, and José González. The Design and Performance of a Conflict-Avoiding Cache. In *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO)*, pages 71–80, 1997.

[87] Pedro Trancoso, Despo Othonos, and Artemakis Artemiou. Data Parallel Acceleration of Decision Support Queries using Cell/BE and GPUs. In *Proceedings of the 6th Conference on Computing Frontiers (CF)*, pages 117–126, 2009.

[88] Cole Trapnell and Michael C. Schatz. Optimizing Data Intensive GPGPU Computations for DNA Sequence Alignment. *Parallel Computing*, 35(8-9):429–440, 2009.

[89] Bin Wang, Yizheng Jiao, Weikuan Yu, Xipeng Shen, Dong Li, and Jeffrey S. Vetter. A Versatile Performance and Energy Simulation Tool for Composite GPU Global Memory. In *Proceedings of IEEE 21st International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 298–302, 2013.

[90] Bin Wang, Zhuo Liu, Xinning Wang, and Weikuan Yu. Eliminating intra-warp conflict misses in GPU. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 689–694, 2015.

[91] Bin Wang, Bo Wu, Dong Li, Xipeng Shen, Weikuan Yu, Yizheng Jiao, and Jeffrey S. Vetter. Exploring Hybrid Memory for GPU Energy Efficiency Through Software-hardware Co-design. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 93–102, 2013.

[92] Bin Wang and Weikuan Yu. Performance and Power Simulation for Versatile GPGPU Global Memory. In *Proceedings of the 27th IEEE International Parallel & Distributed Processing Symposium (IPDPS), Workshops and Phd Forum*, pages 2254–2257, 2013.

[93] Bin Wang, Weikuan Yu, Xian-He Sun, and Xinning Wang. DaCache: Memory Divergence-Aware GPU Cache Management. In *Proceedings of the 29th International Conference on Supercomputing (ICS)*, pages 128–139, 2015.

[94] Teng Wang, Sarp Oral, Michael Pritchard, Bin Wang, and Weikuan Yu. TRIO: Burst Buffer Based I/O Orchestration. In *Proceedings of IEEE International Conference on Cluster Computing (Cluster)*, 2015.

[95] Xinning Wang, Bin Wang, Zhuo Liu, and Weikuan Yu. Preserving Row Buffer Locality for PCM Wear-Leveling Under Massive Parallelism. In *Proceedings of IEEE 23rd International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2015.

[96] Yandong Wang, Yizheng Jiao, Cong Xu, Xiaobing Li, Teng Wang, Xinyu Que, Cristian Cira, Bin Wang, Zhuo Liu, Bliss Bailey, and Weikuan Yu. Assessing the Performance Impact of High-Speed Interconnects on MapReduce. In *Third Workshop on Big Data Benchmarking (WBDB)*, volume 8163 of *Lecture Notes in Computer Science*, pages 148–163, 2012.

[97] Haicheng Wu, Gregory Frederick Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. Kernel Weaver: Automatically Fusing Database Primitives for Efficient GPU Computation. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 107–118, 2012.

[98] Xiaolong Xie, Yun Liang, Guangyu Sun, and Deming Chen. An Efficient Compiler Framework for Cache Bypassing on GPUs. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pages 516–523, 2013.

[99] Yuejian Xie and Gabriel H. Loh. PIPP: Promotion/Insertion Pseudo-partitioning of Multi-core Shared Caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, pages 174–183, 2009.

[100] Yulong Yu, Weijun Xiao, Xubin He, He Guo, Yuxin Wang, and Xin Chen. A Stall-Aware Warp Scheduling for Dynamically Optimizing Thread-level Parallelism in GPGPUs. In *Proceedings of the 29th International Conference on Supercomputing (ICS)*, pages 128–139, 2015.

[101] Yongpeng Zhang, Frank Mueller, Xiaohui Cui, and Thomas Potok. Data-Intensive Document Clustering on Graphics Processing Unit (GPU) Clusters. *Journal of Parallel and Distributed Computing*, 71(2):211–224, February 2011.

[102] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. A Permutation-based Page Interleaving Scheme to Reduce Row-buffer Conflicts and Exploit Data Locality. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture (MICRO)*, pages 32–41, 2000.

[103] Zhong Zheng, Zhiying Wang, and Mikko Lipasti. Adaptive Cache and Concurrency Allocation on GPGPUs. *Computer Architecture Letters*, 6, 2014.