2016

Reducing Cache Contention On Gpus

Kyoshin Choo
University of Mississippi

REDUCING CACHE CONTENTION ON GPUS

A Dissertation
presented in partial fulfillment of requirements
for the degree of Doctor of Philosophy
in the Department of Computer and Information Science
The University of Mississippi

by

Kyoshin Choo

August 2016

ABSTRACT

The usage of Graphics Processing Units (GPUs) as an application accelerator has become increasingly popular because, compared to traditional CPUs, they are more cost-effective, their highly parallel nature complements a CPU, and they are more energy efficient. With the popularity of GPUs, many GPU-based compute-intensive applications (a.k.a., GPGPUs) present significant performance improvement over traditional CPU-based implementations. Caches, which significantly improve CPU performance, are introduced to GPUs to further enhance application performance. However, the effect of caches is not significant for many cases in GPUs and even detrimental for some cases. The massive parallelism of the GPU execution model and the resulting memory accesses cause the GPU memory hierarchy to suffer from significant memory resource contention among threads.

One cause of cache contention arises from column-strided memory access patterns that GPU applications commonly generate in many data-intensive applications. When such access patterns are mapped to hardware thread groups, they become memory-divergent instructions whose memory requests are not GPU hardware friendly, resulting in serialized access and performance degradation. Cache contention also arises from cache pollution caused by lines with low reuse. For the cache to be effective, a cached line must be reused before its eviction. Unfortunately, the streaming characteristic of GPGPU workloads and the massively parallel GPU execution model increase the reuse distance, or equivalently reduce reuse frequency of data. In a GPU, the pollution caused by a large reuse distance data is significant. Memory request stall is another contention factor. A stalled Load/Store (LDST) unit does not execute memory requests from any ready warps

in the issue stage. This stall prevents the potential hit chances for the ready warps.

This dissertation proposes three novel architectural modifications to reduce the contention: 1) *contention-aware selective caching* detects the memory-divergent instructions caused by the column-strided access patterns, calculates the contending cache sets and locality information and then selectively caches; 2) *locality-aware selective caching* dynamically calculates the reuse frequency with efficient hardware and caches based on the reuse frequency; and 3) *memory request scheduling* queues the memory requests from a warp issuing stage, frees the LDST unit stall and schedules items from the queue to the LDST unit by multiple probing of the cache. Through systematic experiments and comprehensive comparisons with existing state-of-the-art techniques, this dissertation demonstrates the effectiveness of our aforementioned techniques and the viability of reducing cache contention through architectural support. Finally, this dissertation suggests other promising opportunities for future research on GPU architecture.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my wife, Jihyun, for fully and pleasantly supporting me with great patience and believing what I am passing through after I quit my job. I could not have done any of this long journey without her. Second, I would like to thank my children for their trust, understanding and always being with me to remind of what is most important. My final personal thanks go to my parents and parents-in-laws for praying, always supporting me in every aspect and encouraging me not to give up and to achieve my goal.

Professionally, I would like to thank my advisor Professor Byunghyun Jang for guiding me with the right direction and pushing me to pursue high-quality research, giving me the freedom to explore my own ideas and helping me develop them. His guidance over the last four years has been invaluable and has lead to the goal. Special thanks to David Troendle for being a great colleague. David's depth of industrial knowledge and insightful thought has helped all the research I have done. I also thank all of the HEROES (HEteROgEneous Systems research) lab members, Tuan Ta, Esraa Abdelmageed, Chelsea Hu, Ajay Sharma, Andrew Henning, Md. Mainul Hassan, Oreva Addoh, Mengshen Zhao, Leo Yi, Michael Ginn, Blake Adams, and Sampath Gowrishetty for their help and invaluable discussion and comments. I would also like to thank Professor Jang's former colleagues, Dana Schaa and Rafael Ubal for giving insightful comments on the simulator.

I would also list to thank the committee members of my PhD qualifying exam and final PhD dissertation defense: Professor Philip J. Rhodes, Professor Feng Wang and Professor Robert J. Doerksen for their direction and suggested improvements to my dissertation. Last but not least, I would like to thank all the faculty members of Computer and Information Science department

at the University of Mississippi: Professor Dawn E. Wilkins, Professor H. Conrad Cunningham, Professor Yixin Chen, Professor J. Adam Jones, Professor P. Tobin Maginnis, former Professor Jianxia Xue, Ms. Kristin Davidson, and Ms. Cynthia B. Zickos for their academic contribution and friendly support.

To my wife Jihyun and our children, Esther, Samuel and Gracie.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

GPU	Graphics Processing Unit
GPGPU	General Purpose Graphics Processing Unit
SIMT	Single-Instruction, Multiple-Thread
SIMD	Single-Instruction, Multiple-Data
MIMD	Multiple-Instruction, Multiple-Data
APU	Accelerated Processing Unit
AMD	Advanced Micro Devices
IC	Integrated Circuit
CPU	Central Processing Unit
CMP	Chip-MultiProcessor
PCI/PCIe	Peripheral Component Interconnect / Peripheral Component Interconnect express
CUDA	Compute Unified Device Architecture
OpenCL	Open Computing-Language
API	Application Programming Interface
DLP	Data-Level Parallelism
TLP	Thread-Level Parallelism
ILP	Instruction-Level Parallelism
CU	Compute Unit
SM	Streaming Multiprocessor
PC	Program Counter
LDST	Load Store

MACU	Memory Access Coalescing Unit
MSHR	Miss Status Holding Register
L1D	Level 1 Data
L1I	Level 1 Instruction
L1T	Level 1 Texture
L1C	Level 1 Constant
L2	Level 2
LLC	Last Level Cache
MC	Memory Controller
FR-FCFS	First-Ready First-Come-First-Serve
PKI	Per Kilo (Thousand) Instructions
MPKI	Misses Per Kilo (Thousand) Instructions
DRAM	Dynamic Random Access Memory
GTO	Greedy-Then-Oldest
LRR	Loose Round Robin
CCWS	Cache-Conscious Warp Scheduling

CHAPTER 1

INTRODUCTION

The success of General-Purpose computing on Graphics Processing Unit (GPGPU) technology has made high performance computing affordable and pervasive in platforms ranging from workstations to hand-held devices. Its cost-effectiveness and power efficiency are unprecedented in the history of parallel computing. Following their success in general-purpose computing, GPUs have shown significant performance improvement in many compute-intensive scientific applications, such as geoscience [79], molecular dynamics [4], DNA sequence alignment [84], and large graph processing [47], physical simulations in science [62], and so on. Furthermore, the advent of the big data era has further stimulated the need to leverage the massive computation power of GPUs in accelerating emerging data-intensive applications, such as data warehousing applications [7, 29, 30, 36, 83, 89] and big data processing frameworks [35, 15, 16, 34, 78]. The GPU-based implementations can provide an order of magnitude performance improvement over traditional CPU-based implementations [36, 83].

Since GPUs were originally introduced to efficiently handle computer graphics workloads in specialized applications such as computer-generated imagery and video games, these traditional GPU workloads involved a large amount of data streaming. To deliver high throughput, GPUs rely on massive fine-grained multithreading to hide long latency operations such as global memory accesses [22], which operates by issuing instructions from different threads when the threads being executed are stalled. Given enough threads and enough memory bandwidth, the GPU's data path can be kept busy, increasing overall throughput at the expense of a single-thread latency.

GPU memory systems have grown to include a multi-level cache hierarchy with both hardware and software controlled cache structures. For instance, Nvidia Fermi GPUs introduced a relatively large (up to 48 KB per core) and configurable L1 cache and 768 KB L2 cache [64]. Likewise, AMD GCN GPUs also offer a 16 KB L1 cache per core and 768 KB L2 cache [2].

While a throughput processor's cache hierarchy exploits application inherent locality and increases the overall performance, the massively parallel execution model of GPUs suffers from the resource contention. In particular, for applications whose performances are sensitive to caching efficiency, such cache resource contention degrades the effectiveness of caches in exploiting locality, thereby suffering from significant performance drop. Experiments show that caches are not always beneficial to GPU applications [41]. From a suite of 12 Rodinia benchmark [14] applications running on a Nvidia Tesla C2070 GPU with its 16 KB L1 caches turned on and off, only two of them show substantial performance improvements from caching, but two show non-trivial performance degradation. The effect of cache is negligible in other 8 benchmarks [41].

One of the main reasons for the performance degradation is cache contention in the memory hierarchy, especially in the L1 data (L1D) cache. The limited resources that serve the massively parallel memory requests destroy the application's inherent memory access locality and create severe cache contention. Software-based optimization could mitigate this problem, but the effort required to revise the existing code to use cache efficiently is non-trivial [24]. A couple of warp schedulers are also proposed to reduce the contention by limiting the active warps in an SM. However, these approaches reduce the utilization of the GPUs [49, 72, 57].

This dissertation proposes architectural support to reduce the cache contention. In particular, this dissertation identifies the causes of the contention and resolves it by detecting the memory access patterns that significantly destroys the memory access locality, calculating reuse frequency of the memory accesses, and probing cache under memory request stall. This dissertation also demonstrates the effectiveness of these proposed techniques by extensively comparing them with

closely related state-of-the-art techniques.

1.1 Research Challenges

One cause of the cache contention arises from *column-strided memory access patterns* that GPU applications commonly generate in many data-intensive applications. Such access patterns are compiled to warp instructions whose generated memory accesses are not well coalesced. We call these instructions *memory-divergent instructions*. The resulting memory accesses are serially processed in the LDST unit due to the resource contention, stress the L1D caches, and result in serious performance degradation. Even though the impact of memory divergence can be alleviated through various software techniques, architectural support for memory divergence mitigation is still highly desirable because it eases the complexity in the programming and optimization of GPU-accelerated data-intensive applications [72, 73].

Cache contention also arises from *cache pollution* caused by low reuse frequency data. For the cache to be effective, a cached line must be reused before its eviction. But, the streaming characteristic of GPGPU workloads and the massively parallel GPU execution model increases the reuse distance, or equivalently reduces the reuse frequency of data. If the low reuse frequency data is cached, it can evict the high reuse frequency data in the cache before it is reused. In a GPU, the pollution caused by a low reuse frequency (i.e., large reuse distance) data is significant.

Memory request stall is another contention factor. A stalled LDST unit does not execute memory requests from any ready warps in the previous issue stage. The warp in the LDST unit retries until the cache resource becomes available. During this stall, the private L1D cache is also in a stall, so no other warp requests can probe the cache. However, there may be data in an L1D cache which may become a hit if other ready warps in the issue stage access the cache. The current structure of the issue stage and L1D unit execution do not allow the provisional cache probing in such a case. This stall prevents the potential hit chances for the ready warps.

1.2 Research Contribution

In this dissertation, we have thoroughly investigated three architectural challenges that can severely impact the performance of GPUs and eventually degrade overall performance. For each challenge, this dissertation proposes solutions to reduce the cache contention such as contention-aware selective caching, locality-aware selective caching, and memory request scheduling. We compare the proposed solutions with the closely related state-of-the-art techniques. In particular, this dissertation has made the following contributions.

- Using the application inherent memory access locality classification along with limitation of resources, we classify cache miss contention into 3 categories, intra-warp (IW), cross-warp (XW), and cross-block (XB) contention according to the cause of the misses. We also classify the cache resource contention into 3 categories, LineAlloc fail, MSHR fail, and MissQueue fail.

- We identify and quantify the factors of the contention such as a column-strided pattern and its resulting memory-divergent instruction, cache pollution by low reuse frequency data, and memory request stall.

- We propose a mechanism to detect the column-strided pattern and its resulting memory-divergent instruction which generates divergent memory accesses, calculate the contending cache sets and locality information, and caches selectively. We demonstrate that the contention-aware selective caching can improve the performance more than 2.25x over baseline and reduce memory accesses.

- We propose a mechanism with low hardware complexity to detect the locality of memory requests based on per-PC reuse frequency and cache selectively. We demonstrate that it improves the performance by 1.39x alone and 2.01x along with contention-aware selective

caching over baseline, prevents 73% of the no-reuse data from caching and improves reuse frequency in the cache by 27x.

- We propose a memory request schedule queue that holds ready warps' memory requests and a scheduler to effectively schedule them to increase the chances of a hit in the cache lines. We demonstrate that there are 12 ready warps on average when the LDST unit is in a stall and this potential improves the overall performance by 1.95x and 2.06x along with contention-aware selective caching over baseline.

1.3 Organization

The rest of this dissertation is organized as follows:

- Chapter 2 details the summary of terminology used in this dissertation, programming model in GPUs, the baseline GPU architecture, and warps scheduling and memory access handling.
- Chapter 3 discusses the data locality, cache miss contention classification, cache resource contention classification and the cache contention factors.
- Chapter 4 presents a contention-aware selective caching proposal to reduce intra-warp associativity contention caused by memory-divergent instructions.
- Chapter 5 presents a locality-aware selective caching to measure the reuse frequency of the instructions and prevent low reuse data from caching.
- Chapter 6 presents a memory request scheduling to better utilize the cache resource under LDST unit stall.
- Chapter 7 discusses related works.
- Chapter 8 concludes the dissertation and discusses the directions for the future work.

CHAPTER 2

BACKGROUND

This chapter introduces background knowledge that serves as a foundation for the rest of this dissertation. In particular, Section 2.1 summarizes the terminology usage in all chapters throughout this dissertation. Section 2.2 describes the GPU programming flow and work-item formation from the software point of view. Section 2.3 presents the abstract model of GPU architecture used in this dissertation. Section 2.4 and Section 2.5 explain how the warps are assigned to the execution units and how global memory requests are handled in the GPU memory hierarchy. More detailed background knowledge can be found in many other references [3, 66, 37, 52].

2.1 Summary of Terminology Usage

This Dissertation	CUDA [66]	OpenCL [52]
thread	thread	work-item
warp	warp	wavefront
thread block	thread block	work-group
SIMD lane	CUDA core	processing element
Streaming Multiprocessor (SM)	SM	Compute Unit (CU)
private memory	local memory	private memory
local memory	shared memory	local memory
global memory	global memory	global memory

Table 2.1. GPU hardware and software terminology comparison between standards.

Table 2.1 summarizes the terminology used in this dissertation. We present this summary to avoid confusion between multiple equivalent technical terms from multiple competing GPGPU

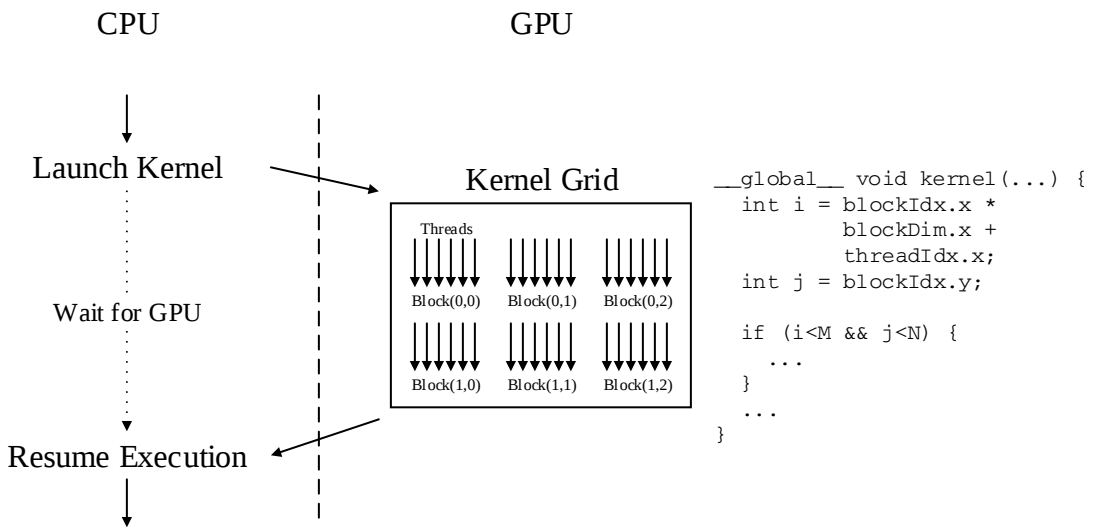


Figure 2.1. A GPU kernel execution flow example - A GPU kernel is launched in host CPU, run on GPU, and then returned to the host CPU. An example CUDA code is shown on the right side.

programming frameworks and industry standards. Detailed definitions of the terms are presented in the following sections. A more thorough explanation of GPU terminologies other than the explanations introduced here can be found in various programming guides and framework specifications [51, 52, 66].

2.2 Programming GPUs

Programming environments and abstractions have been introduced to help software developers write GPU applications. Nvidia's CUDA [66] and Khronos Group's OpenCL [51, 52] are popular frameworks. Despite the terminology differences summarized in Table 2.1, these frameworks have very similar programming models, as introduced below.

Depending on the system configuration, a GPU can be connected via a PCI/PCIe bus or reside on the same die with the CPU. Applications programmed in high-level programming languages, such as CUDA [66] and OpenCL [51, 52], begin execution on CPUs. The GPU portion of the code is launched from the CPU code in the form of *kernels*. Depending on the system, data to be used by a GPU are transferred through the PCI/PCIe bus through an internal interconnection

between the CPU and GPU, or through pointer exchange.

A CUDA or OpenCL program running on a host CPU contains one or more kernel functions. These kernels are invoked by the host program, offloaded to a GPU device, and executed there. The kernel code specifies operations to be performed by the GPU from the perspective of a single GPU *thread*. At kernel launch, the host specifies the total number of threads executing the kernel and their thread grouping. As shown in Figure 2.1, a kernel divides its work into a grid of identically sized *thread blocks*. The size of a thread block is the number of threads (e.g., 256) in that block. From the programmer's perspective, every instruction in the kernel is concurrently executed by all threads in the same thread block. However, there is a limited number of hardware lanes (e.g., 32) that an SM can execute concurrently on real hardware. Consequently, threads are executed in groups of hardware threads called *warps*. The number of threads in a warp is called the size of the warp.

GPU threads have access to various memory spaces. A thread can access its own *private memory*, which is not accessible by other threads. Threads in the same block can share data and synchronize via *local memory*, and all threads in a kernel can access *global memory*. The local and global memories support atomic operations. Global memory is cached in on-chip private L1 data (L1D) cache and shared L2 cache. Much of this dissertation focuses on reducing contention on this L1D cache.

Finally, because every thread executes the same binary instructions as other threads in the same kernel, it must use its thread ID and thread block ID variables to determine its own identity and operate on its own data accordingly. Those IDs can be one- or multi-dimensional, and they are represented by consecutive integer values in each dimension of the block starting with the first dimension, as shown in Figure 2.1.

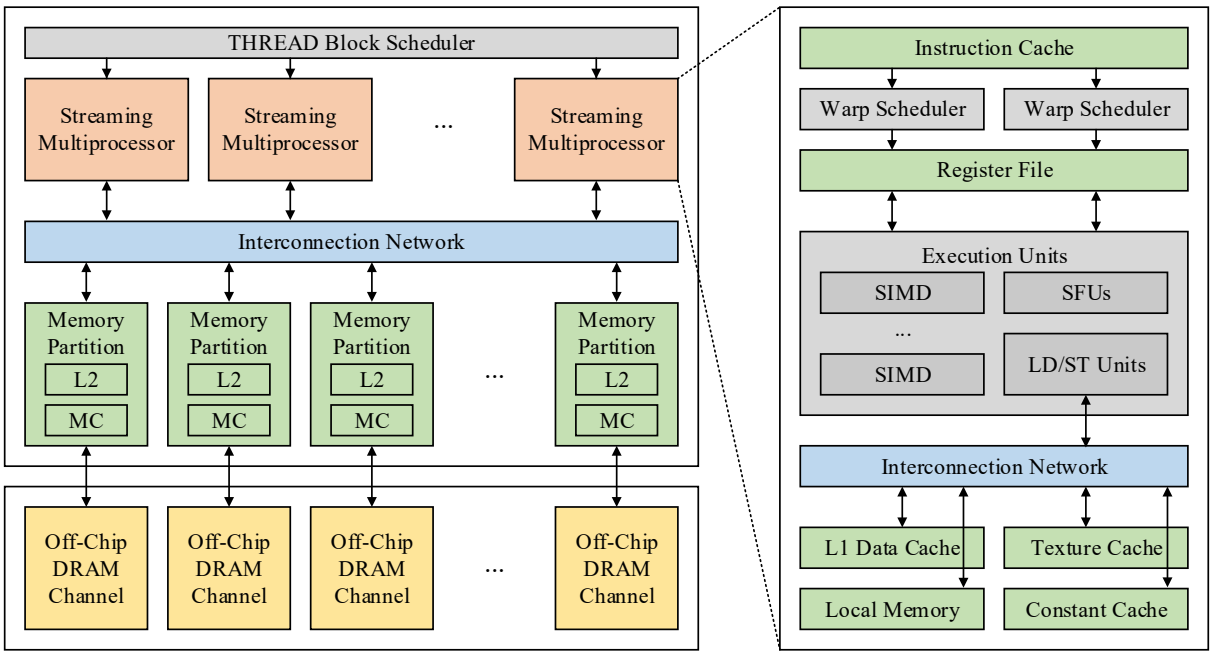


Figure 2.2. Baseline GPU architecture.

2.3 Abstract of GPU Architecture

The modern GPU architecture is illustrated in Figure 2.2. A GPU consists of a thread block scheduler, an array of *Streaming Multiprocessors (SMs)*, an interconnection network between SMs and memory modules, and global memory units. Off-chip DRAM memory (global memory) is connected to the GPU through a memory bus. Each SM is a highly multi-threaded and pipelined SIMD processor. *SIMD lanes* execute distinct threads, operate on a large register file, and progress in lock-step with other threads in the SIMD thread group (warp).

The SM architecture is detailed in the right side of Figure 2.2. It consists of warp schedulers, a register file, SIMD lanes, Load/Store(LDST) units, various on-chip memories including L1D cache, local memory, texture cache, and constant cache. LDST units manage accesses to various memory spaces. Depending on the data being requested, GPU memory requests are sent to either L1D cache, local memory, texture cache or constant cache. Each memory partition consists of L2 cache and a memory controller (MC) that controls off-chip memory modules. An intercon-

Number of SMs	15
SM configuration	1400Mhz, SIMD Width: 16 Warp size: 32 threads Max threads per SM: 1536 Max Warps per SM: 48 Warps Max Blocks per SM: 8 blocks
Warp schedulers per core	2
Warp scheduler	Greedy-Then-Oldest (GTO), default [73, 72] Loose Round-Robin (LRR) Two-Level (TL)[63]
Cache/SM	L1 Data: 16KB 128B-line/4-way (default) L1 Data: 16KB 128B-line/2-way L1 Data: 16KB 128B-line/8-way Replacement Policy: Least Recently Used (LRU) Shared memory: 48KB
L2 unified cache	768KB, 128B line, 16-way
Memory partitions	6
Instruction dispatch throughput per scheduler	2 instructions per cycle
Memory Scheduler	Out of Order (FR-FCFS)
DRAM memory timing	t_{CL}=12, t_{RP}=12, t_{RC}=40, t_{RAS}=28, t_{RCD}=12, t_{RRD}=6
DRAM bus width	384 bits

Table 2.2. Baseline GPGPU-Sim configuration.

nection network handles data transfer between the SMs and L2 caches, and the memory controller handles data transfer between L2 and off-chip memory modules.

We use GPGPU-Sim [5] for detailed architectural simulation of the GPU architecture. The details of the architectural specification can be found in the GPGPU-Sim 3.x manual [82]. We present the details of our simulation setup and configurations in Table 2.2.

2.4 Warp Scheduling

As shown in Figure 2.2, each SM contains multiple physical warp lanes (SIMD) and two warp schedulers, independently managing warps with even and odd warp identifiers. In each cycle, both warp schedulers pick one ready warp and issue its instruction into the SIMD pipeline back-end [64, 65]. 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. 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. 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 [63, 44], warp age [73, 72], instruction age [12], or other statistics that can maximize resource utilization or minimize stalls in memory hierarchy or in execution units. For example, the Greedy-Then-Oldest (GTO) scheduler [73, 72] maintains the highest priority for the currently prioritized warp until it is stalled. The scheduler then selects the oldest among ready warps for scheduling. GTO is a commonly used scheduler because of its good performance in a large variety of general purpose GPU benchmarks.

2.5 Modern GPU Global Memory Accesses

2.5.1 Memory Hierarchy

An SM contains physical memory units shared by all its concurrently executing threads. Private memory mapped to registers is primarily used for threads to store their individual states and contexts. Local memory mapped to programmer-managed scratch-pad memories [8] is used to share data within a thread block. All cores share a large, off-chip DRAM to which global memory maps. Modern GPUs also have a two-level cache hierarchy (L1D and L2) for global memory

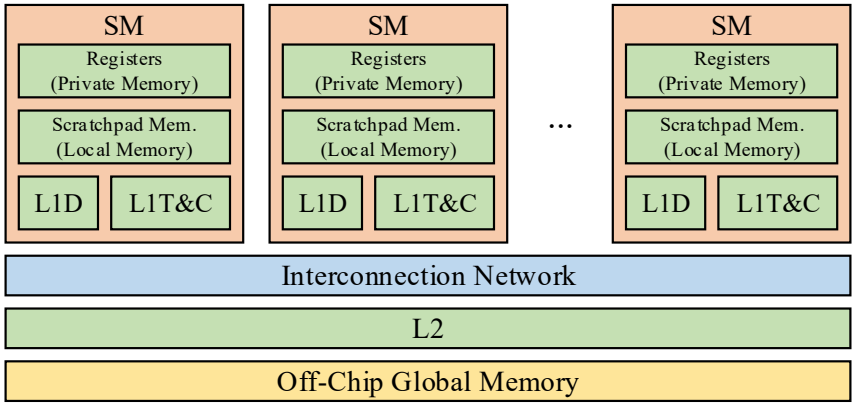


Figure 2.3. A typical memory hierarchy in the baseline GPU architecture. L1D, L1T, and L1C stand for L1 data, L1 texture, and L1 constant caches, respectively.

accesses. Texture and constant caches (L1T and L1C) have existed since the early graphics-only GPUs [33]. Figure 2.3 shows the described memory hierarchy in the baseline GPU architecture.

Current Nvidia GPUs have configurable L1D caches whose size can be 16, 32, or 48 KB. The L1D cache in each core shares the same total 64 KB of memory cells with local memory, giving users dynamically configurable choices regarding how much storage to devote to the L1D cache versus the local memory. AMD GPU L1D caches have a fixed size of 16 KB. Current Nvidia GPUs have non-configurable 768 KB - 1.5 MB L2 caches, while AMD GPUs have 768 KB L2 caches. L1T and L1C are physically separate from L1D and L2, and they are only accessed through special constant and texture instructions. GPU programmers are usually encouraged to declare and use local memory variables as much as possible because the on-chip local memories have a much shorter latency and much higher bandwidth than the off-chip global memory. Small, but repeatedly used data are good candidates to be declared as local memory objects. However, because local memories have a limited capacity and a GPU core cannot access another core's local memory, many programs cannot use local memories due to their data access patterns. In these situations, the global memory with its supporting cache hierarchy should be the choice.

Threads from the same thread block must execute on the same SM in order to use the SM's scratch-pad memory as a local memory. However, when thread blocks are small, multiple thread

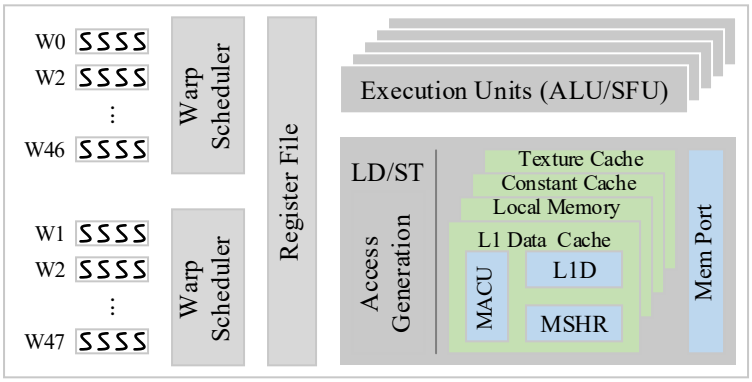


Figure 2.4. A detailed memory hierarchy view.

blocks may execute on a single SM as long as core resources are sufficient. Specifically, four hardware resources - the number of thread slots, the number of thread block slots, the number of registers, and the local memory size - dictate the number of thread blocks that can execute concurrently on a SM; we call this number a SM's (thread) block concurrency. An interconnection network connects SMs to the DRAM via a distributed set of L2 caches and memory modules. A certain number of memory modules share one memory controller (MC) that sits in front of them and schedules memory requests. This number varies from one system to the other. The DRAM scheduler queue size in each memory module impacts the capacity to hold outstanding memory requests.

2.5.2 Memory Access Handling

Figure 2.4 shows a typical thread processing and memory hierarchy on an SM. An instruction is fetched and decoded for a group of threads called a *warp*. The size of warp can vary across devices, but is typically 32 or 64. All threads in a warp execute in an SIMD fashion with each thread using its unique thread ID to map to its data. A user-defined thread block is composed of multiple warps. At issue stage, a warp scheduler selects one of the ready warps for execution.

Figure 2.5 shows the detailed global memory access handling. A memory instruction is issued on a per warp-basis with usually 32 threads in Nvidia Fermi, or 64 threads in AMD Southern

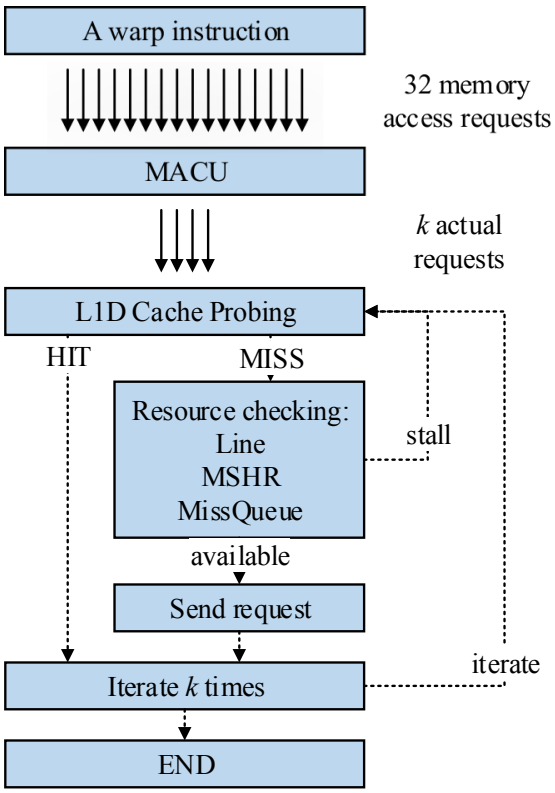


Figure 2.5. Memory access handling procedure.

Island architectures. Once a memory instruction for global memory is issued, it is sent to a Memory Access Coalescing Unit (MACU) for memory request generation to the next lower layer of the memory hierarchy. To minimize off-chip memory traffic, the MACU merges simultaneous per-thread memory accesses to the same cache line. Depending on the stride of the memory addresses among threads, the number of resulting memory requests varies. For example, when 4 threads in a warp access 4 consecutive words, i.e., stride-1 access, in a cache line-aligned data block, the MACU will generate only one memory access to L1D cache as shown in Figure 2.6a. Otherwise, simultaneous multiple accesses are coalesced to a smaller number of memory accesses to L1D cache to fetch all required data. In the worst case, the 4 memory accesses are not coalesced at all and generate 4 distinct memory accesses to L1D cache as shown in Figure 2.6c. Therefore, a fully memory-divergent instruction can generate as many accesses as the warp size. The *working set* is defined as the amount of memory that a process requires in a given time interval [20]. When

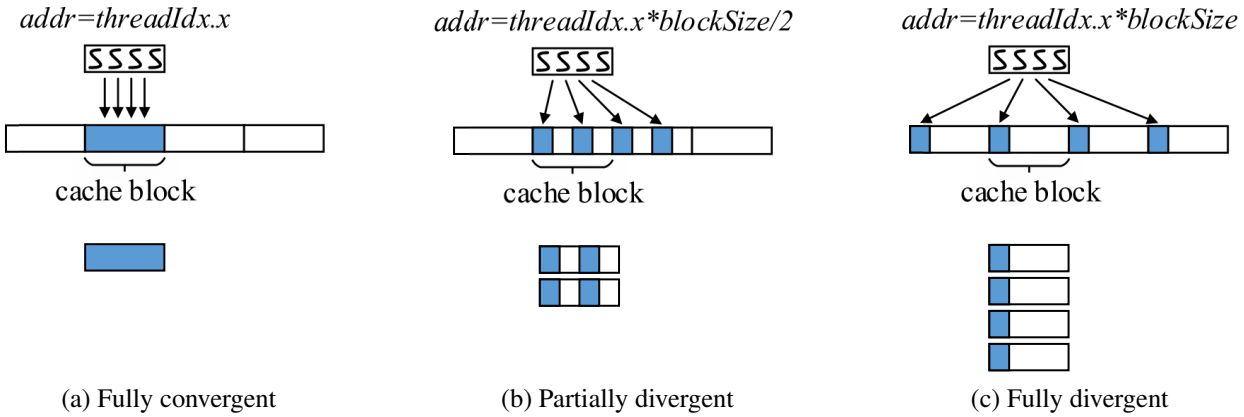


Figure 2.6. Coalescing examples of memory-convergent and memory-divergent instructions.

many memory-divergent instructions are issuing memory requests, the working set size becomes large. If it exceeds the cache size, it causes cache contention. In the rest of this dissertation, the memory instructions that generate only one memory access after MACU are called *memory-convergent instructions*, and the others are called *memory-divergent instructions*. The resultant memory accesses from an MACU are sequentially sent to L1D via a single 128-byte port [12].

When a load memory request hits in L1D, the requested data is written back to the register file and dismissed. If it misses in L1D, the request checks if it can allocate enough resources to process the request. When the request acquires resources, it is sent to the next lower memory hierarchy to fetch data. Otherwise, it retries at next cycle to acquire the resources. The resources to be checked by the request are a line in a cache set, a Miss Status Holding Register (MSHR) entry and a miss queue entry. An allocate-on-miss policy cache allocates one cache line in the destination set of cache if available. Otherwise, it fails allocation on the cache and retries at the next cycle until the resource is ready. An allocate-on-fill policy cache skips this process and attempts allocation on reception of the requested data from the lower memory hierarchy. The MSHR is used to track in-flight memory requests and merge duplicate requests to the same cache line. Upon MSHR allocation, a memory request is buffered into the memory port for network transfer. An MSHR entry is released 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.

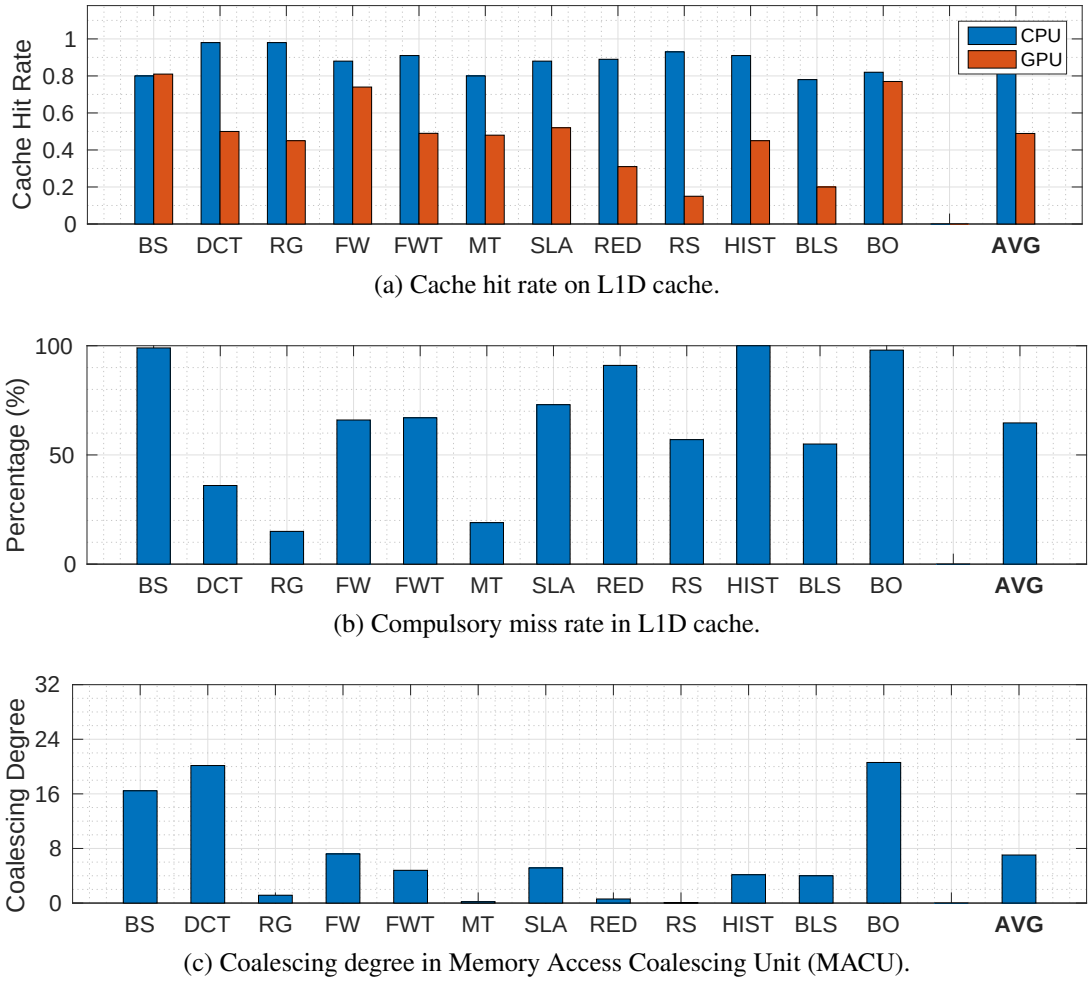
Since L1D caches are not coherent across cores, every global memory store is treated as a write-through transaction followed by invalidation on the copies in the L1D cache [2]. Store instructions require no L1D cache resources and are directly buffered into the memory port to the L1 cache. For this reason, only global memory loads, not stores, are taken into consideration in this dissertation.

2.5.3 Memory Access Characteristics

Since GPUs have a different programming model and execution behavior from traditional CPUs, their memory access also has unique characteristics different from traditional processors. According to our evaluation and analysis of cache behavior and performance [19], GPU memory access has the following characteristics.

Considerably low cache hit rate in L1D: As shown in Figure 2.7a, L1D cache hit rate on GPUs is considerably lower than those on the CPUs (49% vs. 88% on average). This suggests the reuse rate of the data in the cache is not high. This low cache hit rate in L1D results in increased memory traffic to the lower levels of the memory hierarchy, L2 and off-chip global memory. It leads to overall longer memory latency, and thus degrades the overall memory performance.

Compulsory misses dominate: The main cause of the low cache hit rate for L1D results from the fact that compulsory misses dominate in an L1D cache. Compulsory miss is sometimes referred as cold miss or first reference miss since this miss occurs when the data block is brought to the cache for the first time. From Figure 2.7b, the compulsory miss rate over total misses are about 65% on average. This behavior contradicts the conventional wisdom that compulsory misses are negligibly small on traditional multi-core processors [74]. This difference shows that the data in GPU is not reused much, that is, data reusability is very low. This is different from the CPU



(a) Cache hit rate on L1D cache.

(b) Compulsory miss rate in L1D cache.

(c) Coalescing degree in Memory Access Coalescing Unit (MACU).

Figure 2.7. GPU memory access characteristics.

memory access patterns where temporal and spatial localities are used for caching. Tian et al. [80] show that zero-reuse blocks in the L1 data cache are about 45% on average GPU applications.

Coalescing occurs massively: As described in Section 2.2, GPUs achieve high performance through massively parallel thread execution. Such massive thread execution subsequently issues many memory requests to its private L1D cache. Since typical memory accesses are known to be regular with strided patterns in GPUs, massive coalescing occurs at the MACU. Figure 2.7c shows the coalescing degree for each benchmark. The *coalescing degree* is defined as the ratio of the number of memory requests generated by warp instructions to the total L1D cache requests. Since the warp size is 32, maximum coalesce degree is 32. As can be seen in Figure 2.7c, on some

benchmarks such as `bs`, `dct`, and `bo`, the coalescing degree is very high, more than 16. On average, approximately 7 memory requests are coalesced to the same cache line. In other words, the memory traffic reduction by the MACU is 7 to 1.

CHAPTER 3

CACHE CONTENTION

This chapter introduces the taxonomy of memory access locality that GPU applications inherently have. This chapter also introduces two other cache contention classifications, miss contention and resource contention, followed by the factors that are involved in the cache contention. As noted in Section 2.5.2, only global memory loads - not stores - are taken into consideration, since global memory stores bypass the L1D cache and do not affect the cache contention described in this dissertation.

3.1 Taxonomy of Memory Access Locality

A comprehensive understanding of GPU memory access characteristics, especially locality, is essential to a better understanding of contention in the memory hierarchy of a GPU. We introduce the four-category memory access locality including intra-thread locality, intra-warp locality, cross-warp locality, and cross-block locality. The definition of each category follows.

- *Intra-thread data locality (IT)* applies to memory instructions being executed by one thread in a warp. This category captures the temporal and spatial locality of a thread which has a similar pattern to that of the CPU workload.

- *Intra-warp data locality (IW)* applies to memory instructions being executed by threads from the same warp. Depending on the result of coalescing, the instructions are classified to either memory-convergent (in which thread accesses are mapped to the same cache block) or memory-divergent (in which thread accesses are mapped to more than 2 cache blocks).

When the instruction is memory-convergent, the spatial locality between threads is taken care of by the coalescer (MACU), and therefore, the number of memory requests per instruction becomes one, which is the same as the IT locality case. When the instructions are not memory-convergent, each instruction generates multiple memory requests and the working set size becomes larger.

- *Cross-warp Intra-block data locality (XW)* applies to memory instructions being executed by threads from the same thread block, but from different warps in the thread block. If these threads access data mapped to the same cache line, they have XW locality. Warp scheduler and memory latency affect the locality.

- *Cross-warp Cross-block data locality (XB)* applies to memory instructions being executed by threads from different thread blocks, but in the same SMs. If these threads access data mapped to the same cache line, they have XB locality. The thread block scheduler, warp scheduler and memory latency affect the locality. XB locality between thread blocks mapped to different SMs is not considered since they do not show locality in an SM.

Since more threads are involved in the memory access locality for GPUs as described in the taxonomy above, we need to review the definition of temporal and spatial locality. The traditional definition of temporal locality and spatial locality are the followings [37].

Temporal locality: If a particular memory location is referenced at a particular time, then it is likely that the same location will be referenced again in the near future. There is a temporal proximity between the adjacent references to the same memory location. In this case, it is common to make an effort to store a copy of the referenced data in special memory storage, which can be accessed faster. Temporal locality is a special case of spatial locality, namely when the prospective location is identical to the present location.

Spatial locality: If a particular memory location is referenced at a particular time, then

Listing 3.1. SYRK benchmark kernel

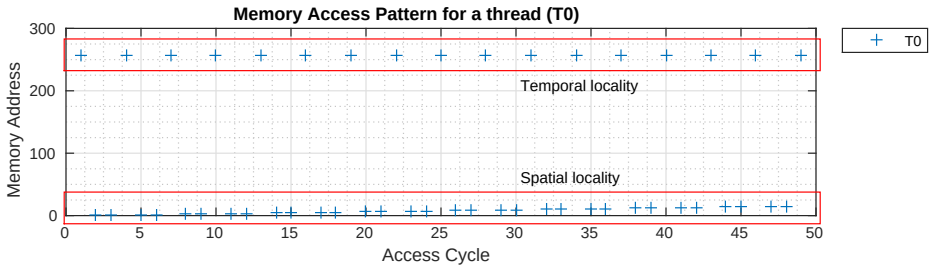
```
__global__ void syrk_kernel(...)
{
   /* C := alpha*A*A' + beta*C , NxN matrix*/
   int j = blockIdx.x * blockDim.x + threadIdx.x;
   int i = blockIdx.y * blockDim.y + threadIdx.y;

   if ((i<N) && (j<N)) {
      c[i*N+j] = c[i*N+j] * beta;
                  ①
      for(int k=0; k<N; k++) {
         c[i*N+j] = alpha * a[i*N+k] * a[j*N+k] + c[i*N+j];
                            ②          ③          ④
      }
   }
}
```

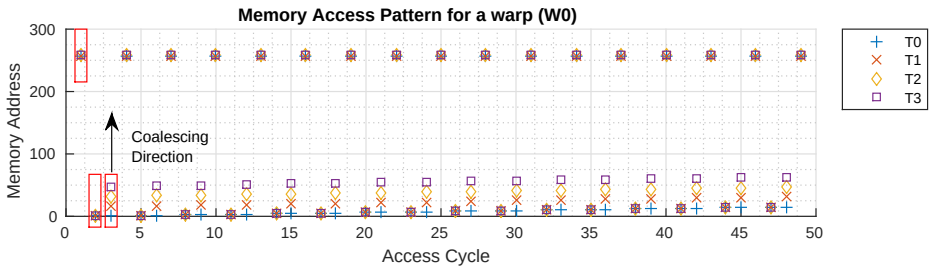
it is likely that nearby memory locations will be referenced in the near future. In this case, it is common to attempt to guess the size and shape of the area around the current reference for which it is worthwhile to prepare faster access.

To visualize the above category in detail, we choose a benchmark, `syrk`, in Polybench/GPU [31]. The `syrk` benchmark has intra-warp (IW) locality as well as cross-warp intra-block (XW) and cross-warp cross-block (XB) locality. The example kernel code in Listing 3.1 and address distribution per category are given. For simplicity in this example, we assume $N = 16$, warp size 4, thread block size 16. There are 16 thread blocks total, and the maximum warps in an SM are 4 thread blocks.

The code listing above shows four load instructions, which are highlighted and labeled in circled numbers in the listing. The load instruction execution sequence is $\{1, \{2, 3, 4\}^N\}$ where $\{...\}^N$ represents repetition of the sequence in the curly bracket and the N value is 16 in this case. Since N is 16, there are 49 load memory accesses per thread. The memory address pattern for the thread 0 ($T0$) is shown in Figure 3.1a, intra-thread locality (IT). Since load instructions 1 and 4 read the same memory address regardless of the k value, these load instructions have temporal locality in a thread memory access. On the other hand, since load instructions 2 and 3



(a) Memory access pattern for a thread ($T0$) - Intra-thread (IT) locality.



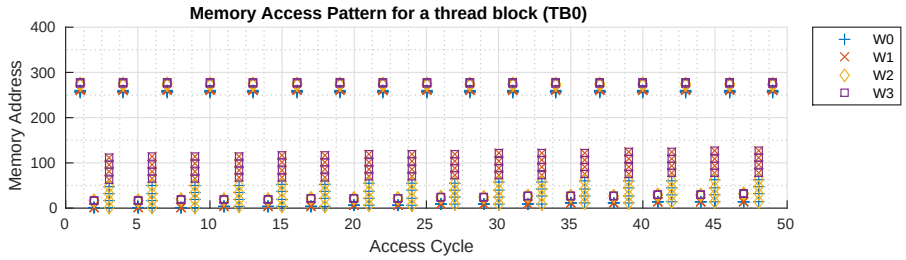
(b) Memory access pattern for a warp ($W0$ with $T0, T1, T2, T3$) - Intra-warp (IW) locality. Memory accesses within a rectangle can be coalesced.

Figure 3.1. Memory access pattern for a thread and a warp.

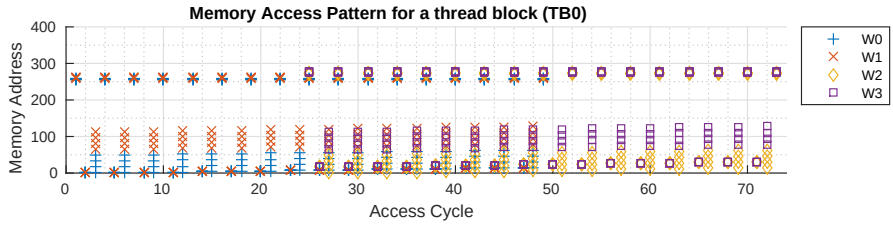
are accessing neighboring memory addresses, respectively, this shows spatial locality. Hence, the memory pattern for the thread T0 shows both temporal and spatial locality.

The next scope is intra-warp locality (IW). Figure 3.1b shows the memory address pattern for the 4 threads, $T0, T1, T2$, and $T3$, in the same warp. Since coalescing occurs at the intra-warp level, as marked with red rectangles in the figure, coalescing occurs within the red rectangle, in the vertical direction of address distribution in a cycle. As explained in Section 2.5.2, only memory addresses within the cache block size are coalesced. Therefore, when the memory requests are scattered enough not to be grouped into a cache line, i.e., N is larger, the memory-divergent instruction generates up to warp-size memory requests. The temporal locality and spatial locality observed in the IT locality graph still hold.

The next scope is cross-warp intra-block (XW) locality as in Figure 3.2. From this scope and beyond, the warp scheduler plays a very important role. The address pattern without the warp scheduler effect is shown in Figure 3.2a. The pairs, $\{W0, W2\}$, and $\{W1, W3\}$ show very similar

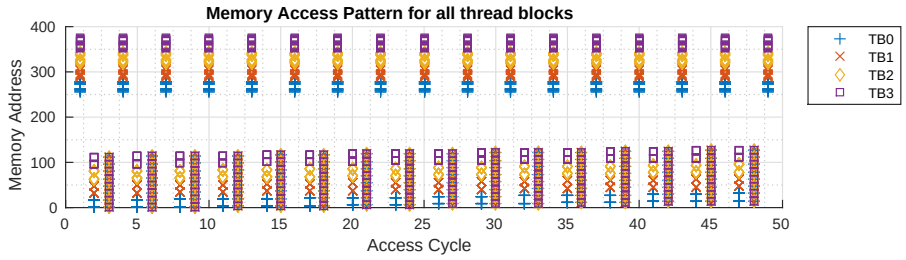


(a) Memory access pattern for a thread block ($TB0$) - ideal - Cross-warp Intra-block (XW) locality.

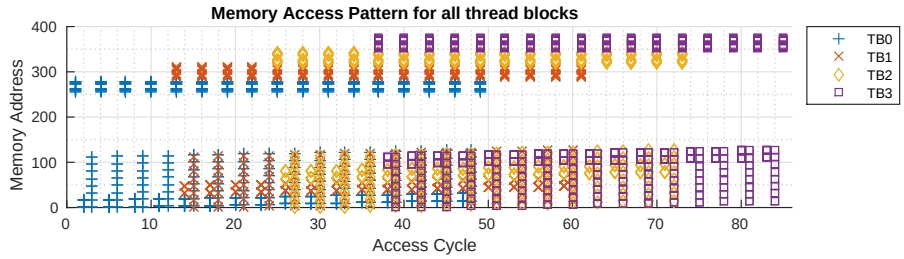


(b) Memory access pattern for a thread block - skewed by warp scheduler effect - Cross-warp Intra-block (XW) locality.

Figure 3.2. Memory access pattern for a thread block.



(a) Memory access pattern for an SM - ideal - Cross-warp Cross-block (XB) locality.



(b) Memory access pattern for an SM - skewed by warp scheduler effect - Cross-warp Cross-block (XB) locality.

Figure 3.3. Memory access pattern for an SM.

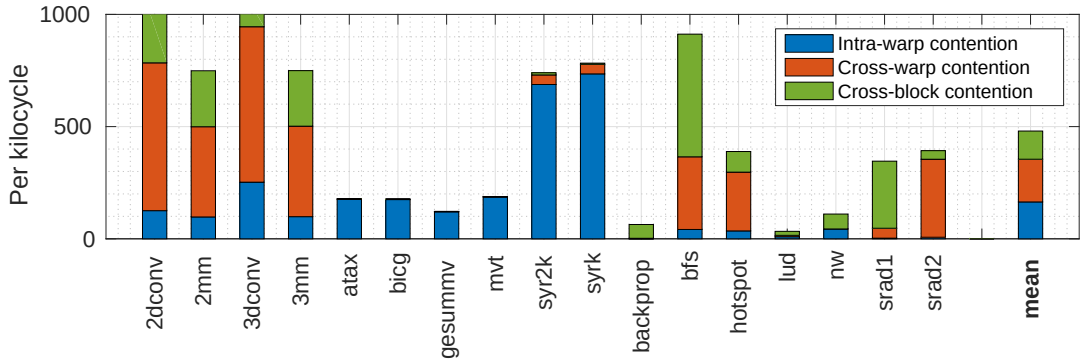
memory access pattern and that pattern has very good spatial and temporal locality. When we include the warp scheduler effect as shown in Figure 3.2b, the memory accesses are more scattered in time. In this figure, by scheduling the locality pairs apart, for example $\{W0, W1\}$ first and $\{W2, W3\}$ later, the locality shown in the code can be completely destroyed. As shown, the x-axis of Figure 3.2b is also extended to 1.5x cycles longer. When it comes to the real warp scheduler introduced in Section 2.4, the memory access pattern is much more complex. As the number of memory requests grows, the chances of thrashing grow high since, in a short period of time, many memory accesses are contending to acquire the cache resources.

The largest scope is cross-warp cross-block (XB) locality. Figure 3.3 shows this case. The working set of memory access grows even larger than in the XW case. Figure 3.3a shows the memory access pattern without the effect of the warp scheduler and the memory latency. This still shows good inter-block locality between the blocks without scheduler effect, while the scheduler effect affects the locality pattern significantly.

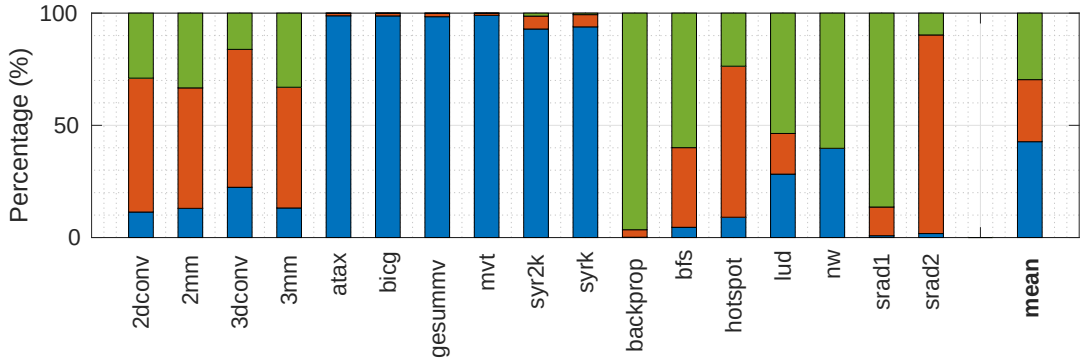
In summary, GPU applications have inherent memory access locality either at the thread level, warp level, cross-warp level or cross-block level. Since threads are executed as a warp in GPUs, the thread level locality is absorbed to the warp level by coalescing at the MACU unit. However, when the memory accesses with locality are executed in a GPU, they are competing with each other to acquire the limited resources such as warp schedulers, LDST units, caches, and other resources in the GPU. When the number of memory accesses at a certain time interval (working set) is large, the contention becomes severe. This contention creates a serious performance bottleneck.

3.2 Taxonomy of Cache Contention

Due to the resource limitations of the memory hierarchy, as introduced in Section 3.1, the inherent memory access locality can be dramatically reduced and causing cache miss contention and cache resource contention. According to the cause of the contentions, we classify them as



(a) Miss contention classification (per kilocycle).



(b) Miss contention classification (percentage).

Figure 3.4. Classification of miss contentions at L1D cache in per kilocycle and in percentage.

follows.

3.2.1 Cache Miss Contention Classification

When a cache miss occurs, the new memory request must evict a previously residing cache line (victim cache line). Depending on the warp ID and block ID relationship between the inserting and evicting lines, the cache miss is classified into intra-warp, cross-warp, and cross-block contention. We classify the GPU cache miss contention as follows:

- *Intra-Warp contention (IW)* refers to contention among memory requests from threads in the same warp. The contention can occur among the memory requests from the same instruction of the same warp or among the memory requests from different instructions of the same warp. We define the former case as *coincident IW* contention because the contention is

between the requests occurring at the same time frame (during coalescing), and the latter case as *non-coincident IW* contention.

- *Cross-Warp Contention (XW)* refers to contention among memory requests from threads in different warps. Since many warps execute concurrently in an SM, the memory working set size from those warps tends to be large. Hence, the cross-warp contention frequently causes capacity misses. Data blocks are evicted frequently before any reuse occurs, especially when the reuse distance is long. More importantly, memory requests with no reuse may evict cache lines that have high reuse, resulting in cache pollution.

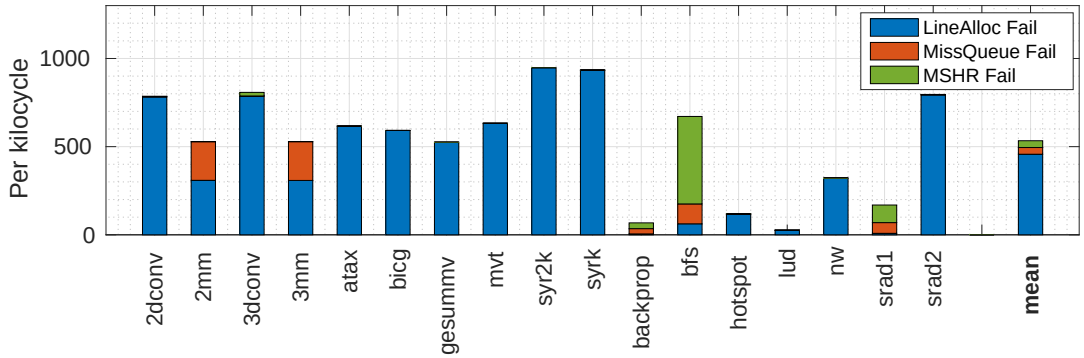
- *Cross-Block Contention (XB)* refers to contention among memory requests from threads in different thread blocks. This is a subset of the XW contention where contention source is from two different blocks.

Figure 3.4 shows the miss contention classification in L1D cache depending on the cause of miss contention. From the figure, about 45% of the cache misses on average are from intra-warp contention. Especially, for the benchmarks such as `atax, bicg, gesummv, mvt, syr2k`, and `syrk`, intra-warp contention dominates the miss contention.

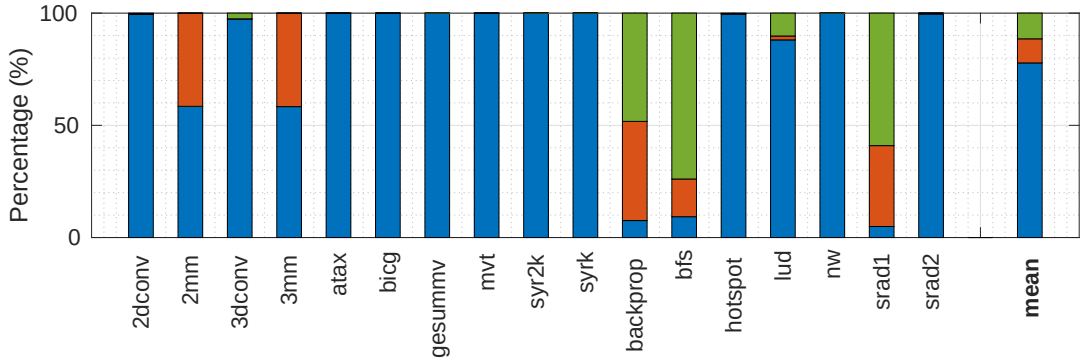
3.2.2 Cache Resource Contention Classification

When an incoming memory access request is serviced in the cache hierarchy after a miss, it checks the cache to see if there are enough resources such as cache line, MSHR entry, and miss queue entry to serve the request. When the cache does not have enough resources, the incoming request is stalled and retries until it acquires the cache resources it needs. Depending on the type of the resource acquisition fail, we classify the resource contention into three categories:

- *Line Allocation Fail* occurs when there is no cache line available for allocation. When a request misses in a cache, the request tries to find an available line in the cache for allocation



(a) Resource contention (per kilocycle).



(b) Resource contention (percentage).

Figure 3.5. Resource contentions at L1D cache in per kilocycle and in percentage.

under *Allocation-on-Miss* policy. When all the lines are reserved, line allocation fail occurs and the request retries until a line is available.

- *MSHR Fail* occurs when the request can reserve a line in a set but there is no entry available in the MSHR. The MSHR holds an active request sent to the lower level cache until the request returns with data.

- *Miss Queue Fail* occurs when a line is available to be allocated and there is a MSHR entry available, but the miss queue to the lower level is full. It also means that the lower level cache is blocked.

The graph in Figure 3.5 shows the metrics for each type of resource reservation fails. Approximately 80% of the resource contention is LineAllocFail and the resource contention takes

Type	Processor	L1 cache	Threads / Core	Cache / Thread	
CPU	Intel Haswell	32 KB	2	16 KB	
	Intel Xeon-Phi	32 KB	4	8 KB	
	AMD Kaveri	64 KB	2	32 KB	
Type	Processor	L1 cache	Threads / Core	Cache / Thread	Cache / Warp
GPU	Nvidia Fermi	48 KB	1536	32 B	1 KB
	Nvidia Kepler	48 KB	2048	24 B	750 B
	AMD SI	16 KB	2560	6.4 B	410 B

Table 3.1. Cache capacity across modern multithreaded processors.

about 500 cycles per kilocycles on average for the benchmarks we tested.

3.3 Cache Contention Factors

3.3.1 Limited Cache Resource

Modern GPUs have widely adopted hardware-managed cache hierarchies inspired by the successful deployment in CPUs. However, traditional cache management strategies are mostly designed for CPUs and sequential programs; replicating them directly on GPUs may not deliver the expected performance as GPUs' relatively smaller cache can be easily congested by thousands of threads, causing serious contention and thrashing.

Table 3.1 lists the L1D cache capacity, thread volume, and per-thread and per-warp L1 cache size for several state-of-the-art multithreaded processors. For example, the Intel Haswell CPU has 16 KB cache per thread per core available, but, the NVIDIA Fermi GPU has only 32 B cache per thread available, which is significantly smaller than CPU cache. Even if we consider the cache capacity per warp (i.e., execution unit in a GPU), it has only 1 KB per warp, which is still far smaller than for the CPU cache. Generally, the per-thread or per-warp cache share for GPUs is much smaller than for CPUs. This suggests the useful data fetched by one warp is very likely

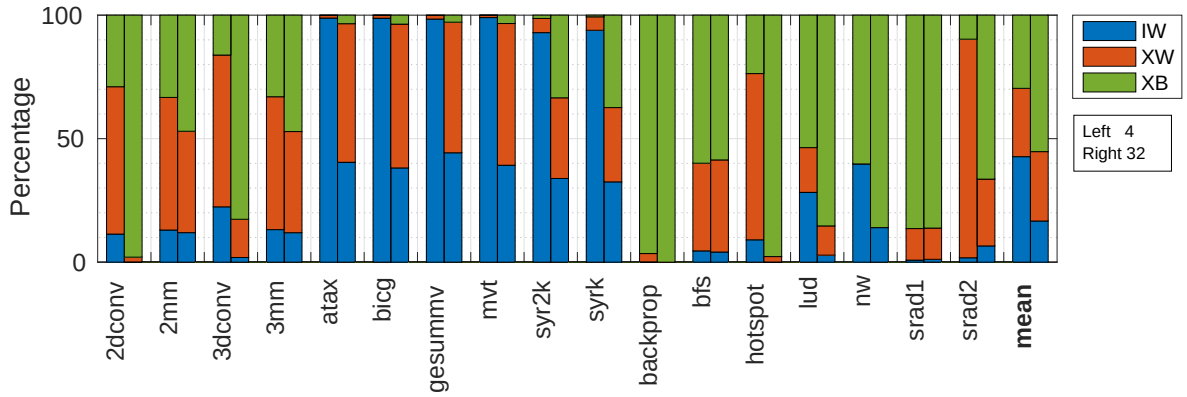


Figure 3.6. Classification of cache misses (intra-warp(IW), cross-warp(XW), and cross-block(XB) miss) and comparison with different associativity (4-way and 32-way) caches. Left bar is with 4-way associativity and right with 32-way.

to be evicted by other warps before actual reuse. Likewise, the useful data fetched by a thread can also be evicted by other threads in the same warp. Such eviction and thrashing conditions destroy locality and impair performance. Moreover, the excessive incoming memory requests can lead to significant delay when threads are queuing for the limited resources in caches (e.g., a certain cache set, MSHR entries, miss buffers, etc.). This is especially so during an accessing burst period (e.g., in the starting phase of a kernel) or set-contending coincident IW contention.

3.3.2 Column-Strided Accesses

The cache contention analysis in Section 3.2 shows that the intra-warp contention takes about 45% of the overall cache miss contention and that 80% of the resource contention is line allocation fail. The analysis infers that the intra-warp associativity contention has a big impact on GPU performance. In order to illustrate the problem of intra-warp contention and quantify their direct impacts on GPU performance, we used two L1D configurations that have the same total capacity (16 KB) but different cache associativities (4 vs 32) to execute 17 benchmarks from PolyBench [31] and Rodinia [14].

As shown in Figure 3.6, after increasing the associativity from 4 to 32, the intra-warp misses in `atax, bicg, gesummv, mvt, syr2k, syrk` are reduced significantly. About

45% of the misses in `gesummv` are still intra-warp (IW) misses, because it has two fully memory divergent loads that contend for the L1D cache. Even though a 32-way cache is impractical for real GPU architectures, this experiment shows that eliminating associativity conflicts are critical for high performance in benchmarks with memory-divergent instructions.

In benchmarks with multidimensional data arrays, the column-strided access pattern is prone to create this 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 the 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 $set = (addr/blkSz)\%nset$, where $addr$ is the target memory address, $blkSz$ is the length of cache line and $nset$ is the number of cache sets. For example, in the Listing 3.1, when the address stride between two consecutive threads is equal to a multiple of $blkSz * nset$, all blocks needed by a single warp are mapped into the same cache set. When the stride size ($STRIDE$) is 4096 bytes as in the $kernel\_1$ below, the 32 consecutive intra-warp memory addresses, 0x00000, 0x01000, 0x02000, ..., 0x1F000, will be mapped into the set 0 in our baseline L1D that has 4-way associativity, 32 cache sets, and 128B cache lines.

Since cache associativity is often much smaller than warp size, 4 (associativity) versus 32 (warp size) in this example, associativity conflict occurs within each single memory-divergent load instruction and then the memory pipeline is congested by the burst of intra-warp memory accesses.

3.3.3 Cache Pollution

While GPGPU applications may exhibit good data reuse, due to the small size of the cache and heavy contention in L1D cache as well as many active memory requests, the distance between those accesses that could exploit reuse effectively becomes farther apart, resulting in a miss. Figure 3.7 shows the distribution of reuse from the execution of each benchmark with 16 KB, 32-set,

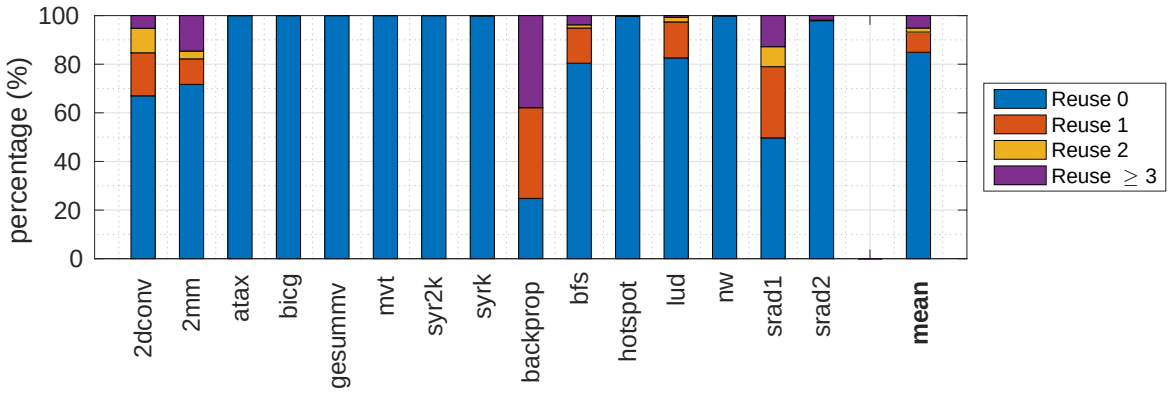


Figure 3.7. Block reuse percentage in the L1D cache. *Reuse0* represents no-reuse until eviction.

4-way, 128-byte block size cache. The value of reuse is defined as the number of accesses of a line after insertion to the eviction. The initial load to the cache line sets the value to zero. Each successive hit of the line increases the reuse value by one. *Reuse0* represents that the line is never reused after its insertion to the cache. As can be seen, the *Reuse0* dominates the distribution with about 85%. That is, many of the cache lines are polluted by the one time use data. If the one time use data is detected before insertion and is not inserted, the efficiency of the cache will increase. Also, if the lines are not polluting the cache, the reuse frequency of other lines would increase.

3.3.4 Memory Request Stall

When the LDST unit becomes saturated, the new request to the LDST unit will not be accepted. When the LDST unit is in a stall by one of the cache resources, for example, a line, an MSHR entry or a miss queue entry, the request fully owns the LDST unit and retries for the resource. Whenever the contending resource becomes free, the retried request finally acquires the resource and the LDST unit can accept the next ready warp. While the stalled request retries, the LDST unit is blocked by the request and cannot be preempted by another request from other ready warps. Usually, the retry time is large because the active requests occupy the resource during the long memory latency to fetch data from the lower level of the cache or the global memory.

During this stall, other ready warps which may have the data they need in the cache cannot

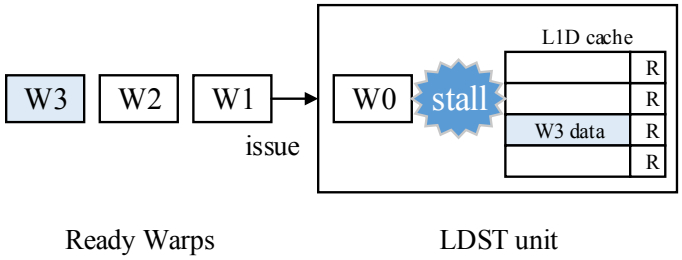


Figure 3.8. LDST unit is in a stall. A memory request from ready warps cannot progress because the previous request is in stall in the LDST unit.

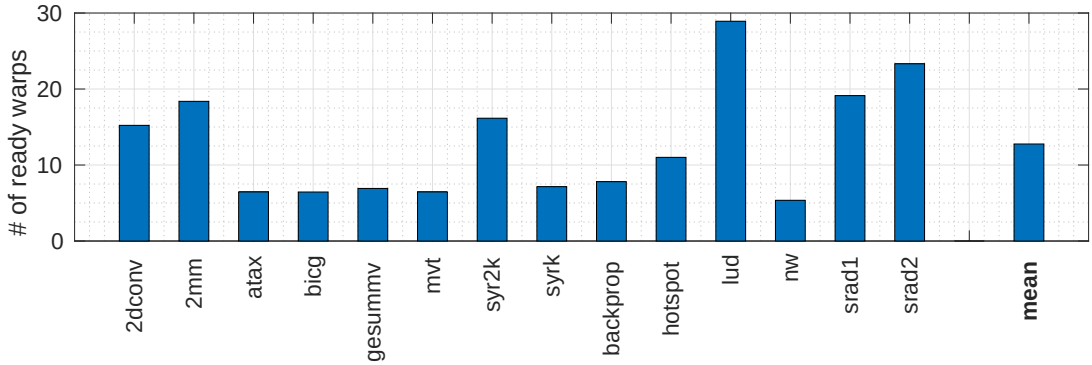


Figure 3.9. The average number of ready warps when cache resource contention occurs.

progress because the LDST unit is in a stall caused by the previous request. This situation is illustrated in Figure 3.8. Assume $W0$ is stalled in the LDST unit and there are 3 ready warps in the issue stage. While $W0$ is in a stall, the 3 ready warps in the issue stage cannot issue any request to the LDST unit. Even though $W3$ data is in the cache at that moment, it cannot be accessed by $W3$. While $W0$, $W1$, and $W2$ are being serviced in the LDST unit, the $W3$ data in the cache may be evicted by the other requests from $W0$, $W1$, or $W2$. When $W3$ finally accesses the cache, the previous $W3$ data in the cache has already been evicted and then the $W3$ request misses in the L1D cache and would need to send a fetch request to the lower level to fetch the data. If $W3$ had a chance to be scheduled to the stalled LDST unit, it would make a hit in the cache and save extra cycles.

To estimate the potential opportunity for the hit under this circumstance, we measured the number of ready warps when the memory request stall occurs. This number does not dictate the

number of hit increase, but it gives us an estimate. Figure 3.9 shows that 12 warps on average are ready to be issued when the LDST unit is in a stall.

CHAPTER 4

CONTENTION-AWARE SELECTIVE CACHING

4.1 Introduction

Our analysis in Chapter 3 shows that if the memory accesses from a warp do not coalesce, L1D cache gets populated fast. The worst case scenario is a column-strided access pattern that maps many accesses to the same cache set. This creates severe resource contention, resulting in stalls in the memory pipeline. Furthermore, the simultaneous memory accesses from several in-flight warps cause contention as well. The widely used associativity of L1D cache is 4-way and it can be significantly smaller compared to the number of per-thread divergent memory accesses. There could be as many as 32 threads generating divergent memory accesses. This contention puts severe stress on the L1 data cache.

To distribute these concentrated accesses across cache sets, memory address randomization techniques for GPU have been proposed [75, 86]. They permute the cache index bits by *logical xoring* to distribute the concentrated memory accesses over the entire cache uniformly. However, dispersion over the entire cache does not work well since there are many in-flight warps and the memory access pattern of the warps are similar. It does not effectively reduce the active working set size.

To mitigate the contention generated by memory divergence, this chapter presents a proactive contention detection mechanism and selective caching algorithm depending on the concentration per cache set measure and a Program Counter (PC)-based locality measure to maximize the benefits of caching. In this chapter, we identify the problematic intra-warp associativity contention

in GPU and analyze the cause of the problem in depth. Then, we present a proactive contention detection mechanism to use when contention-prone memory access pattern occurs. We also propose a selective caching algorithm based on the concentration per cache set measure and per-PC based locality measure. Thorough analysis on the experimental result follows.

4.2 Intra-Warp Cache Contention

4.2.1 Impact of Memory Access Patterns on Memory Access Coalescing

Depending on the stride of the memory addresses among threads, the number of resulting memory requests is determined by the MACU. For example, when 4 threads of a warp access 4 consecutive words (i.e., a stride of 1) in a cache line aligned data block, the MACU will generate only one memory request to L1D cache. We call this case a *memory-convergent instruction* as shown in Figure 4.1a. Otherwise, simultaneous multiple requests are not fully coalesced and generate several memory requests to L1D cache to fetch all demanded data. In the worst case, the 4 memory requests are not coalesced at all and generate 4 distinct memory requests to L1D cache. We call this case a *fully memory-divergent instruction* as shown in Figure 4.1c. If the number of the generated memory requests are in between, we call it a *partially memory-divergent instruction* as shown in Figure 4.1b. We define the number of the resulting memory requests as the *memory divergence degree*.

4.2.2 Coincident Intra-warp Contention Access Pattern

As described in Section 3.2.1, a large portion of the cache contention for the benchmarks are from coincident intra-warp contention. Listing 4.1 shows the `bicg` benchmark kernel code and a column-strided access pattern. The column-major strided access pattern is prone to create this high coincident 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.

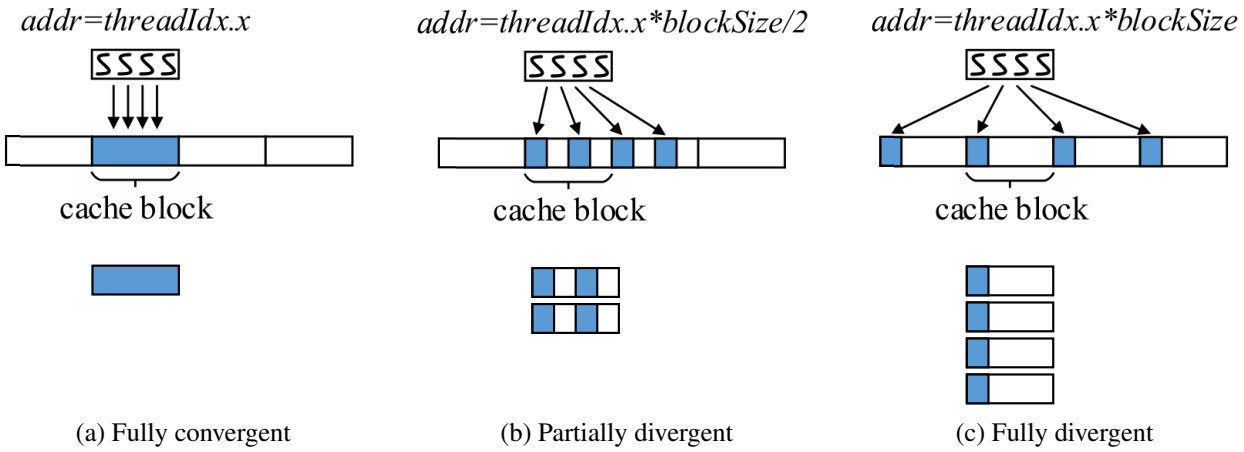


Figure 4.1. (*Revisited*) Coalescing example for memory-convergent instruction and memory-divergent instruction.

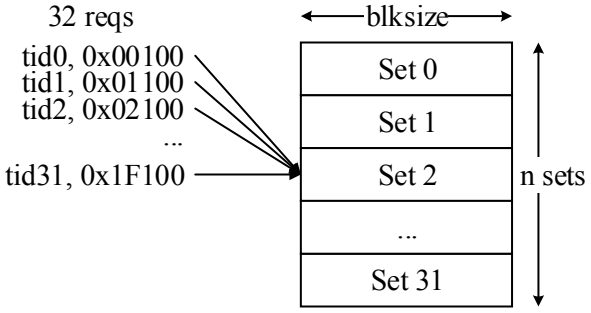
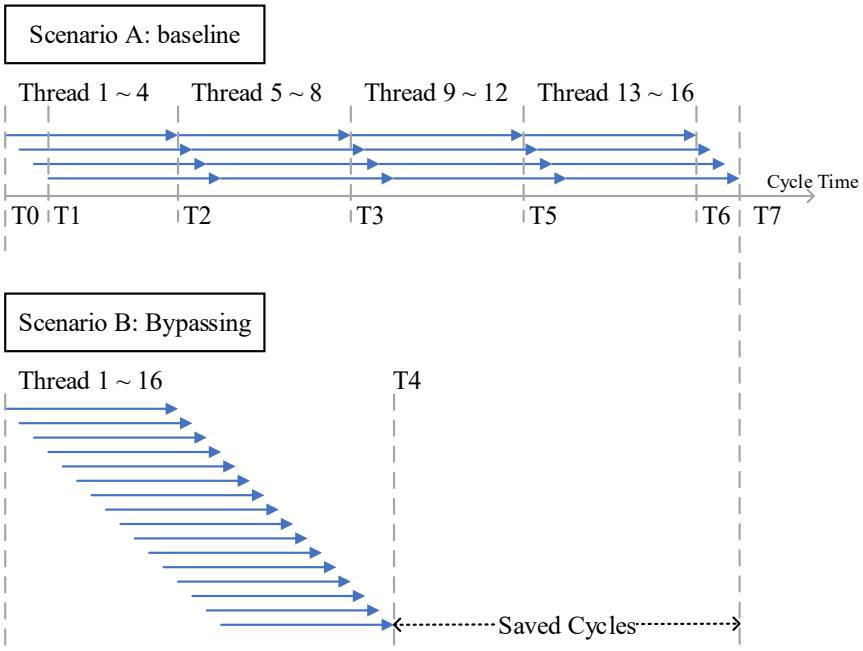


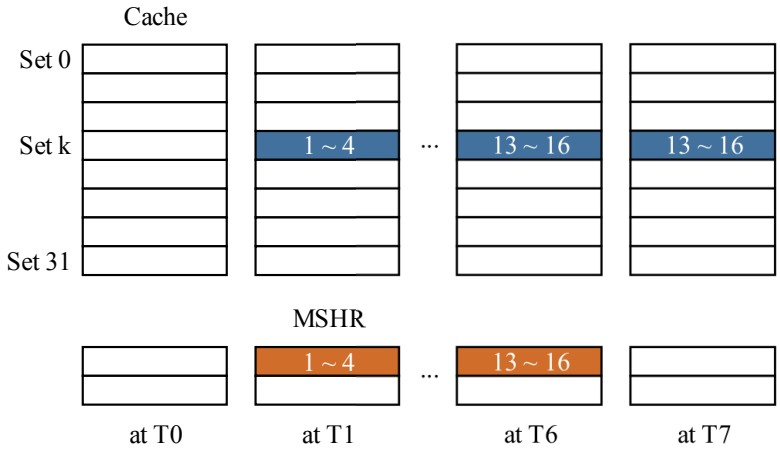
Figure 4.2. Example of contending set by column-strided accesses.

By using this pattern, each thread iterates a stride of data independently. In a conventional cache indexing function, the target set index is computed as $set = (addr/blkSz)\%nset$, where $addr$ is the target memory address, $blkSz$ is the length of cache line and $nset$ is the number of cache sets. From Listing 4.1 and Figure 4.2, when the address stride between two consecutive threads is equal to a multiple of $blkSz * nset$, all blocks needed by a single warp are mapped into the same cache set. When the stride size, $stride$, is 4096 words as in the $kernel\_1$, the 32 consecutive intra-warp memory addresses, 0x00100, 0x01100, 0x02100, ..., 0x1F100, will be mapped into the set 2 in our baseline L1D that has 4-way associativity, 32 cache sets, and 128 B cache lines.

Since cache associativity is often smaller than warp size, 32 in this example, associativity conflict occurs within each single memory-divergent load instruction and then the memory pipeline



(a) Illustration of the cache congestion in L1D cache when column-strided pattern occurs (Scenario A) and the bypassed L1D accesses (Scenario B). Each arrow represents turnaround time of the request.



(b) Cache contents and MSHR contents at $T0$, $T1$, and $T7$

Figure 4.3. Example of BICG memory access pattern.

Listing 4.1. BICG benchmark kernel_1 and kernel_2

```
// Benchmark bicg's two kernels
// Thread blocks: 1-dimensional 256 threads
// A[] is a 4096-by-4096 2D matrix stored
//  as a 1D array
// Code segment is simplified for demo

__global__ bicg_kernel_1 (...) {
   int tid = blkIdx.x * blkDim.x + tIdx.x;
   for (int i = 0; i < NX; i++) // NX = 4096
      s[tid] += A[i * NY + tid] * r[i];
}

__global__ bicg_kernel_2 (...) {
   int tid = blkIdx.x * blkDim.x + tIdx.x;
   for (int j = 0; j < NY; j++) // NY = 4096
      q[tid] += A[tid * NY + j] * p[j];
}
```

is congested by the burst of intra-warp memory accesses. Figure 4.3a Scenario A illustrates the case. When the fully divergent warp instruction is issued, it reaches the MACU. Since it is a fully divergent instruction, it generates as many memory requests as the warp size. Since, as assumed, this instruction has column-strided pattern, the generated cache set indexes are the same, namely *set k*. Cycle $T0$ in Figure 4.3a indicates the beginning of the memory access. The generated memory requests begin sending requests by probing the L1D cache. With the allocation-on-miss policy, the first 4 accesses for threads 1 through 4 acquire cache lines in the set k between $T0$ and $T1$, and send requests to the lower level cache after allocating MSHR entries. The MSHR entry is filled at $T1$. Until the response for the request arrives, the lines are allocated and marked as *reserved*, so that later attempts to acquire the line fails. Since 4 lines are already allocated by the first 4 memory accesses, the remaining requests from 5 through 32 requests should wait until at least one of the responses of the first 4 requests arrives from the lower level. At $T2$, the response for the request of thread 1 arrives, fills the cache and modifies the state of the line to valid. However, as soon as the response is cached, the waiting request evicts the line right away and reserves the line for its request. The final request from the warp memory requests fills the cache line as illustrated

in Figure 4.3b at $T7$.

4.3 Selective Caching

High cache contention on GPUs can change cache behavior by destroying locality and creating contention. Our analysis shows that the majority of cache contention is caused by coincident intra-warp contention. Therefore, we propose a locality and contention aware selective caching that avoids non-beneficial caching. Figure 4.4 shows the overall flow of selective caching proposed in this chapter. It consists of three major components: memory divergence detection, cache index calculation, and locality degree calculation.

4.3.1 Memory Divergence Detection

When a group of memory addresses requested yields a linear access pattern [40], all memory accesses from a warp are coalesced into one memory request by MACU. Otherwise, it generates more than one memory request to the lower level cache. As described in Section 4.2.2, when the memory divergence is greater than the size of the cache associativity, the LDST unit starts to stall to acquire the cache resource as in Figure 4.3a. Therefore, detecting this memory divergence degree plays a key role in avoiding coincident intra-warp contention. We implement this memory divergence detector using a counter in each LDST unit per SM. The counter value for the detector varies from 1 (when fully coalesced) to the SIMD width which is typically 32 (when fully divergent). The memory divergence detection is marked as ② in Figure 4.4.

Once the memory divergence degree is detected, the next step is to decide whether to cache or bypass[1]. As described in Section 4.2.2, when the memory divergence is greater than the size of the cache associativity, n, a stall can start and cause a serious bottleneck. We propose 3 decision schemes when memory divergence is detected. Scheme a is to bypass all the requests (no caching

[1]Bypassing can be turned on and off by flipping a dedicated bit of memory instruction bit stream on most modern GPUs.

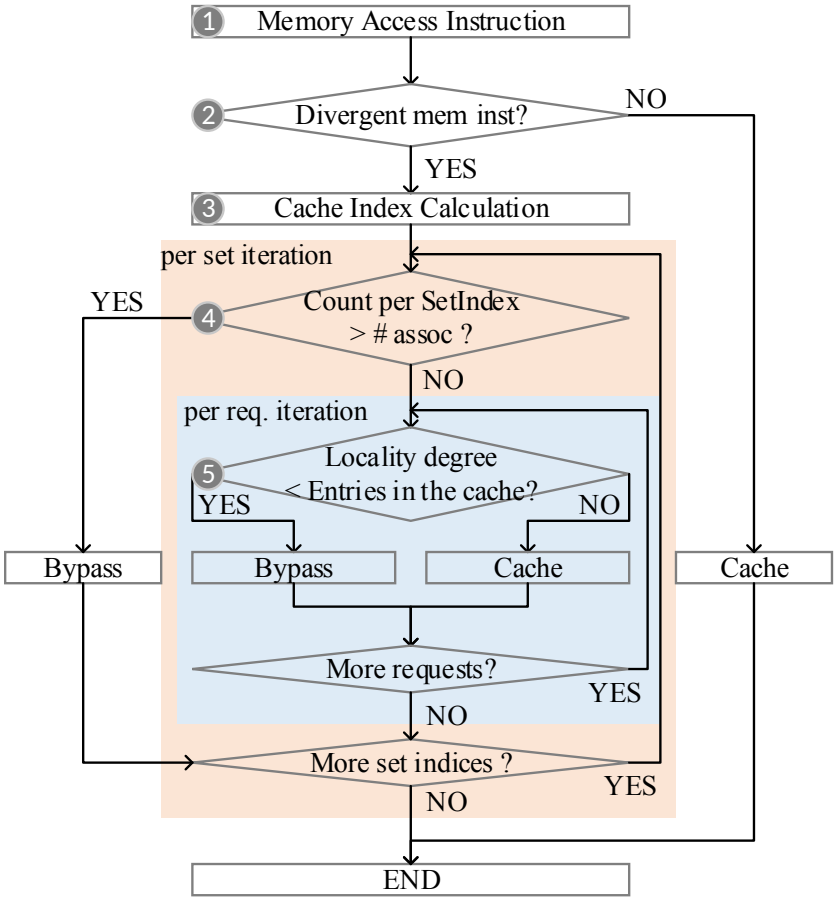


Figure 4.4. The task flow of the proposed selective caching algorithm in an LDST unit.

at all, Figure 4.5a). Scheme b is to partially cache the first n requests and bypass all other requests (Figure 4.5b). The last scheme, scheme c, is to partially cache the last n requests and bypass all other requests (Figure 4.5c). The motivation of schemes b and c is that the instruction can preserve intra-warp or cross-warp locality which may occur by the cached lines in subsequent instruction execution. Scheme c shows the same cache entry of the baseline after executing the instruction as shown in Scenario A of Figure 4.3a.

The example in Figure 4.3b Scenario B illustrates how bypassing the memory requests from the memory-divergent instruction reduces the associativity congestion. Since memory requests from each thread are being bypassed, there is no need to allocate a line in the cache set. Also, the lines already in the cache do not need to be evicted, thus preserving the locality of the previously

allocated lines in the cache. As in Scenario B, when all requests are passed to lower level cache, they do not congest the cache resource. At $T4$, all the responses are returned and the number of cycles, marked as 'Saved Cycles', $T6 - T4$, is reduced as a result of the selective caching.

4.3.2 Cache Index Calculation

When all the generated memory accesses map to the same cache set, the selective caching schemes described in Section 4.3.1 do not cause cache congestion and the schemes a, b, c operate effectively. However, when all the generated memory accesses do not fall into the same cache set, but into a couple of different cache sets, we need to expand the schemes explained in Section 4.3.1 to apply to each of the sets.

We calculate the cache set index for each request and count the number of requests for each set. When the number of requests for a set is greater than the associativity n, then we cache the last n and bypass the others for the set as described in Scheme c in Section 4.3.1. We iterate this for the calculated sets. The cache index calculation is marked as ③ and ④ in Figure 4.4. The portion marked with 'per set iteration' is iterated per set.

Assume that there are 8 memory requests total. After cache index calculation, 5 requests map to the cache set i and 3 requests map to the cache set k in a cache with the associativity of 2 in Figure 4.5d. Since set associativity is 2 in this example, the last 2 out of 5 memory requests for the cache set i are cached and the remaining 3 requests are bypassed. Likewise, the last 2 out of 3 memory requests for the set k are cached and the remaining 1 request is bypassed. Since the cache index mapping logic simply extracts the address bits in the memory request to calculate the cache set index, the additional hardware is negligible.

4.3.3 Locality Degree Calculation

The selective cache scheme with the cache index calculation from Section 4.3.2 always caches as many requests as the associativity size whenever memory-divergent instruction is de-

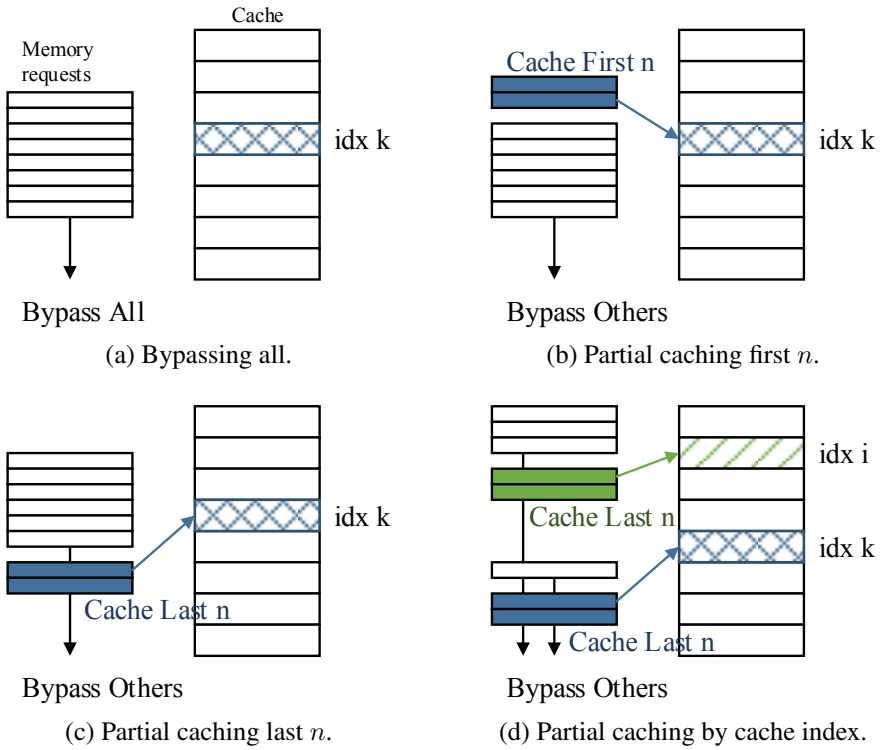


Figure 4.5. Different selective caching schemes with associativity size n when the memory divergence is detected.

tected. When replacing the cache line with new requests, if the evicted line is more frequently or actively used by the same warp or other warps, then it destroys the locality. When finding a victim cache line to be evicted, if we use *locality degree* for each cache line, we can preserve the most frequently used line.

The locality degree calculation is marked as ⑤ in Figure 4.4. The portion marked with 'per req. iteration' is iterated for every request. Locality degree can be calculated by counting the re-references for each instruction. When the locality degree of the current memory request is smaller than the locality degree of any of the existing lines in the destination cache set, then it is better for the new line to bypass the cache. If any of the locality degree of the existing lines is smaller than the locality degree of the current memory request, then the line is evicted and the current memory request is cached. This strategy keeps frequently-used cache lines and preserves discovered locality.

To implement this locality degree feature, we need to add two things to the baseline, new fields in each cache line and a re-reference table. For the cache lines, a 7-bit PC and a 4 bit re-reference counter are added. For the PC representation, 7 hashed bit is enough to distinguish among PCs for distinct load instructions. Since the number of distinct load instructions in compute kernels are known to be not many, 18 on average over all Polybench [31] and Rodinia [14] benchmarks, 7 hashed bit for the PC is enough. The re-reference table contains two fields 7-hashed-bit PC and the average re-reference counter. Whenever a hit occurs in a cache, the counter is incremented by one. Whenever a line is evicted, the PC and re-reference counter pair is added to the re-reference table. When added to the table, the re-reference counter is added as a running sum.

When a request needs to be cached after memory divergence detection and cache index calculation, the average re-reference counter of the new request indexed by its PC is compared with the re-reference counter of the lines in the cache set. If the re-reference counter of the new request is higher than any others in the cache set, then a victim line is selected and the new request is cached into the victim line. Otherwise, it is bypassed.

4.4 Experiment Methodology

4.4.1 Simulation Setup

We configured and modified GPGPU-Sim v3.2 [82], a cycle-accurate GPU architecture simulator to find contention in current GPU memory hierarchy, and implemented the proposed algorithms. The NVIDIA GTX480 hardware configuration is used for the system description. The baseline GPGPU-Sim configurations for this chapter are summarized in Table 4.1.

4.4.2 Benchmarks

To perform our evaluations, we chose benchmarks from the Rodinia [14] and PolyBench/GPU [31]. We pruned our workload list by omitting the applications provided in the benchmark for

Number of SMs	15
SM configuration	1400 Mhz, SIMD Width: 16 Warp size: 32 threads Max threads per SM: 1536 Max Warps per SM: 48 Warps Max Blocks per SM: 8 blocks
Warp schedulers per core	2
Warp scheduler	Greedy-Then-Oldest (GTO) [73] (default) Loose Round-Robin (LRR) Two-Level (TL)[63]
Cache/SM	L1 Data: 16 KB 128B-line/4-way (default) L1 Data: 16 KB 128B-line/2-way L1 Data: 16 KB 128B-line/8-way Rep. Policy: Least Recently Used (LRU) Shared memory: 48 KB
L2 unified cache	768 KB, 128 B line, 16-way
Memory partitions	6
Instruction dispatch throughput per scheduler	2 instructions per cycle
Memory Scheduler	Out of Order (FR-FCFS)
DRAM memory timing	t_{CL}=12, t_{RP}=12, t_{RC}=40, t_{RAS}=28, t_{RCD}=12, t_{RRD}=6
DRAM bus width	384 bits

Table 4.1. Baseline GPGPU-Sim configuration.

basic hardware profiling and graphics interoperability. We also omitted kernels that did not have a grid size large enough to fill all the cores and whose grid size could not be increased without significantly changing the application code. The benchmarks tested are listed in Table 4.2.

4.5 Experimental Results

4.5.1 Performance Improvement

Figure 4.6a shows the normalized IPC improvement of our proposed algorithm over the baseline. The baseline graph is shown with a normalized IPC of 1. *Assoc-32* has the same size

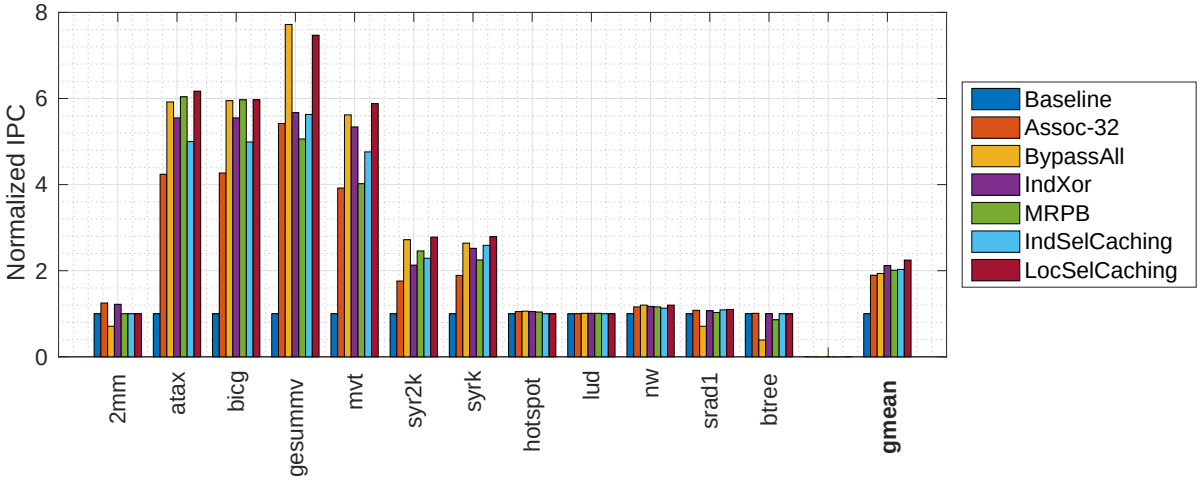
Name	Suite	Description
2MM	PolyBench	2 Matrix Multiplication
ATAX	PolyBench	Matrix Transpose and Vector Mult.
BICG	PolyBench	BiCG SubKernel of BiCGStab Linear Sol.
GESUMMV	PolyBench	Scalar, Vector and Matrix Multiplication
MVT	PolyBench	Matrix Vector Product and Transpose
SYR2K	PolyBench	Symmetric rank-2k operations
SYRK	PolyBench	Symmetric rank-k operations
HotSpot	Rodinia	Hot spot physics simulation
LUD	Rodinia	LU Decomposition
NW	Rodinia	Needleman-Wunsch
SRAD1	Rodinia	Speckle Reducing Anisotropic Diffusion
BTREE	Rodinia	B+tree

Table 4.2. Benchmarks from PolyBench [31] and Rodinia [14].

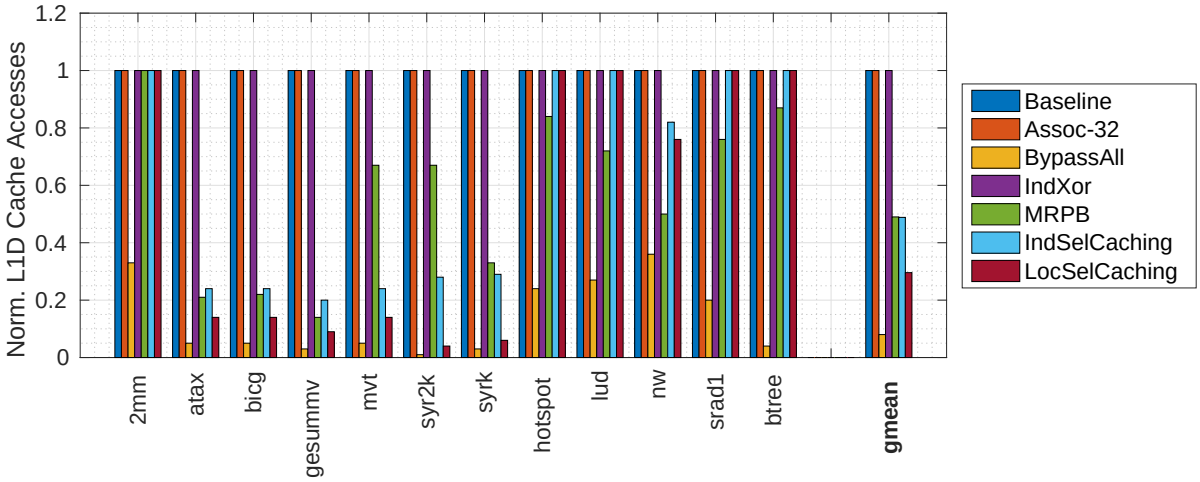
cache but 32-way associativity. *BypassAll* bypasses all the incoming load instructions. *IndSel-Caching* represents the cache index calculation selective caching. *LocSelCaching* represents the locality degree based selective caching. For performance comparison with other techniques to reduce intra-warp contention, we chose an alternative indexing scheme, *IndXor*, [86] and memory request prioritization, *MRPB* [42].

The benchmarks that have severe coincident intra-warp contention such as `atax, bicg, gesummv, mvt, syr2k, syrk` in Figure 3.4 show significant IPC improvement. For other benchmarks that do not suffer much from intra-warp contention, the selective caching algorithm does not provide much benefit. The average IPC improvement is calculated using the geometric mean. Note that the simple bypassing algorithm, *BypassAll*, outperforms the baseline with the given benchmark sets, confirming that GPGPU applications suffer from unnecessary caching. Our proposed scheme, *LocSelCaching*, outperforms the baseline by 2.25x, and outperforms the *IndXor* and *MRPB* by 9% and 12%, respectively.

Figure 4.6b shows the reduction of L1D cache accesses. *BypassAll*, as expected, reduces



(a) IPC improvement.



(b) L1D cache access reduction.

Figure 4.6. Overall improvement - IPC improvement and L1D cache access reduction.

the L1D cache accesses the most[2]. However, its performance is not better than other schemes because simple bypassing does not take advantage of locality. *IndXor* does not reduce the L1D cache access at all since it only changes the cache indexing scheme to reduce intra-warp contention. *MRPB* and *IndSelCaching* reduce similar amounts of L1D cache traffic. *LocSelCaching* reduces about 71% of the L1D cache accesses. The cache access reduction makes direct positive impact on power consumption, average latency, and finally the overall performance.

[2]Since we do not exclude the write accesses, even the *BypassAll* case has L1D accesses. The L1D cache access reduction graph contains write cache accesses to show the overall reduction rate.

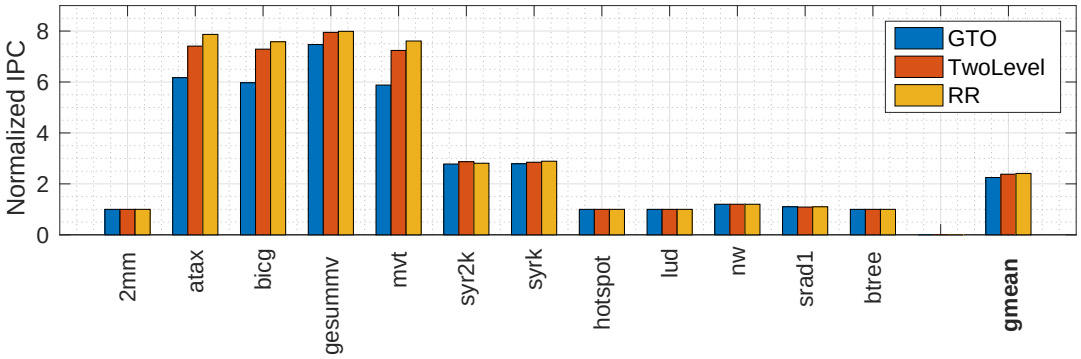


Figure 4.7. IPC improvement for different schedulers.

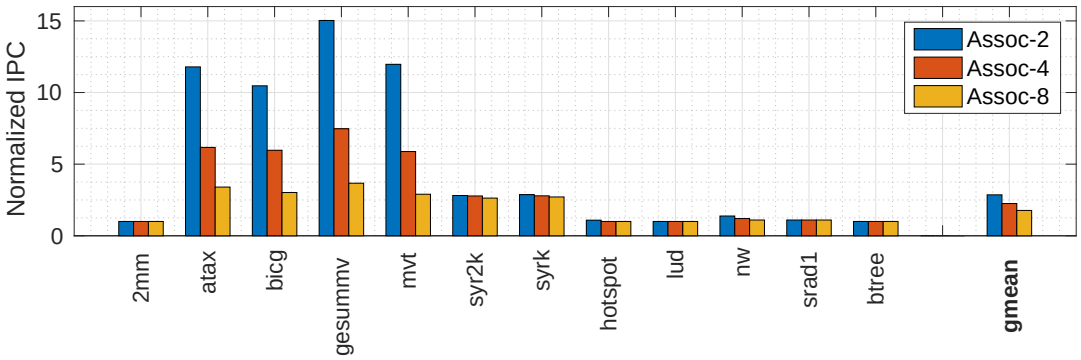


Figure 4.8. IPC improvement for different associativities.

4.5.2 Effect of Warp Scheduler

Our experiment described in Section 4.5.1 uses the *Greedy-Then-Oldest (GTO)* warp scheduler. Since our selective caching algorithm reduces the coincident intra-warp contention, other schedulers should not affect the trend of overall performance improvement. Figure 4.7 shows the result with different warp schedulers, such as *TwoLevel* [63] and basic *RoundRobin* warp scheduler. Since *GTO* is designed to reduce cross-warp contention and *TwoLevel* is designed to improve core utilization by considering branch diversity, *GTO* scheduler enhancement is the least while the *RR* scheduler enhancement is the most. Overall, experiments demonstrate that the selective caching scheme improves coincident intra-warp contention effectively with different schedulers.

4.5.3 Cache Associativity sensitivity

Figure 4.8 shows the result with different cache associativities: 2-, 4-, and 8-way. Coincident intra-warp contention tends to be much more severe with a smaller associativity cache since the associativity to warp size ratio is even worse. According to the analysis shown in Figure 4.3a, memory requests become more serialized and the stall becomes worse than for other associativity cases. Therefore, the IPC improvement for associativity 2 is the most among the three different associativity cases. Cache with associativity 8 still improves the performance by about 1.77x.

4.6 Related Work

4.6.1 Cache Bypassing

Jia et al. [41] evaluated GPU L1D cache locality in current GPUs. To justify the effect of the cache in GPUs, they showed a simulation result with and without L1D cache. Then, the authors classified cache contentions into three categories: within-warp, within-block, and program-wide. Based on the category, they proposed compile-time methods, and their proposed compile-time methods analyze GPU programs and determine if caching is beneficial or detrimental and apply it to control the L1D cache to bypass or not.

Jia et al., in their MRPB paper [42] showed an improved algorithm to bypass L1D cache accesses. When any of the resource unavailable events happen that may lead to a pipeline stall, the memory requests from the L1D cache are bypassed until resources are available. They also prioritize memory accesses what occurred from cross-warp contention to minimize cross-warp contention. They discovered that the massive memory accesses from different warps worsen the memory resource usage. Therefore, instead of sending the memory requests from the SMs to L1D cache directly, they introduce a buffer to rearrange the input requests and prioritize per-warp accesses and reduce stalls in a memory hierarchy. This technique reduces cross-warp contention; however, without cooperation with the warp scheduler, the effect is not significant. Their bypassing

technique is triggered when it detects cache contention. While it reacts to the contention, our algorithm proactively detects cache contention in advance by detecting memory divergence and selectively caches depending on the locality information.

The most recent research by Li et al. [57] also exploits bypassing to reduce contention. They extract locality information during compile time and throttle the warp scheduled to avoid thrashing due to warp contentions. Their work focuses on reducing cross-warp contention.

While the existing works determine bypassing upon occurred contention or static compile time information which may prove to be incorrect at runtime, our work focuses more on proactively detecting and avoiding cache contention. Also, our work incorporates dynamically obtained locality information to preserve locality.

4.6.2 Memory Address Randomization

Memory address randomization techniques for CPU caches are well studied. Pseudo-random cache indexing methods have been extensively studied to reduce conflict misses. Topham et al. [81] use XOR to build a conflict-avoiding cache; Seznec and Bodin [77, 11] combine 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 [54, 69, 70, 91]. Khairy et al. [49] use XOR-based Pseudo Random Interleaving Cache (PRIC) which is used in [81, 70].

Another common approach is to use a secondary indexing method for alternative cache sets when conflicts happen. This category of work includes skewed-associative cache [77], column-associative cache [1], and v-way cache [67].

Some works have also noticed that certain bits in the address are more critical in reducing cache miss rate. Givargis [27] uses off-line 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. [75] propose 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 a direct-mapped cache. ASCIB also needs extra storage to track the changes in the address bits. Wang et al. [86] applied an XOR-based index algorithm in their work. It finds the address range that is more critical to reduce intra-warp cache contention.

The main purpose of the different cache indexing algorithms is to randomly distribute the congested set over all cache sets to minimize the cache thrashing. However, GPGPU executes massively parallelized workloads as much as possible, so requests from different warps can easily overflow the cache set size. For example, NVIDIA Fermi architecture uses 16KB L1 data cache which has 128 cache lines (32 sets with 4 lines per set). Just 4 warp's set contending accesses quickly fill up the cache. Indexing may distribute the cache access throughout the whole cache; however, caching the massive amount of cache accesses which may not be used more than once is polluting cache and evicts cache lines which have better locality.

4.7 Summary

This chapter presented that the massive parallel thread execution of GPUs causes significant cache resource contention that is not sufficient to support massively parallel thread execution when memory access patterns are not hardware friendly. By observing the contention classification for miss and resource contention, we also identified that the coincident intra-warp contention is the main culprit of the contention, and the stall caused by the contention severely impacts the overall performance.

In this chapter, we proposed a locality and contention aware selective caching based on memory access divergence to mitigate coincident intra-warp resource contention in the L1 data (L1D) cache on GPUs. First, we detect memory divergence degree of the memory instruction to determine whether selective caching is needed. Second, we use a cache index calculation to further

decide which cache sets are congesting. Finally, we calculate the locality degree to find a better victim cache line.

Our proposed scheme improves the IPC performance by 2.25x over the baseline. It outperforms the two state-of-the-art mechanisms, IndXor and MRPB, by 9% and 12%, respectively. Our algorithm also reduces 71% of the L1D cache access which results in reducing power consumption.

CHAPTER 5

LOCALITY-AWARE SELECTIVE CACHING

5.1 Introduction

Unlike CPUs, GPUs run thousands of concurrent threads, greatly reducing the per-thread cache capacity. Moreover, typical GPGPU workloads process a large amount of data that do not fit into any reasonably sized caches. Streaming-style data accesses on GPUs tend to evict to-be-referenced blocks in a cache. Such pollution of cache hierarchy by streaming data degrades the system performance.

A simple solution is to increase the cache size. However, this is not a cost effective approach since caches are expensive components. Therefore, a good GPU cache management technique which dynamically detects streaming patterns and selectively caches the memory requests for frequently used data is very desired. A technique that dynamically determines which blocks are likely or unlikely to be reused can avoid polluting the cache. This can make a non-trivial impact on overall performance.

To reduce the contention caused by placing the streaming data into the cache, this chapter presents a locality-aware selective caching mechanism. To track the locality of memory requests dynamically, we propose a hardware efficient *reuse frequency table* which maintains the average reuse frequency per instruction. Since GPUs typically execute a relatively small number of instructions per kernel (i.e., threads execute the same single sequence of instructions), the number of memory instructions are usually quite small. Maintaining the reuse frequency table indexed by a hashed Program Counter (PC) keeps the hardware inexpensive. By carefully investigating the

threshold for a caching decision, we minimize the implementation of the average reuse frequency to a single bit. This chapter makes the following contributions:

- Through the identification of the cache resource contention in GPU cache hierarchy, line allocation fail, MSHR fail and miss queue fail, we identifies that there is a large number of no-reuse blocks polluting cache.
- We propose a hardware efficient locality tracking table, *reuse frequency table*, for dynamically maintaining the average reuse frequency per instruction. We also define the table update procedure.
- We propose a selective caching algorithm based on the dynamic reuse frequency table which has a per-instruction locality measure to effectively identify streaming accesses and bypass.

5.2 Motivation

5.2.1 Severe Cache Resource Contention

To reduce memory traffic and latency, modern GPUs have widely adopted hardware-managed cache hierarchies inspired by their successful deployment in CPUs. However, traditional cache management strategies are mostly designed for CPUs and sequential programs; replicating them directly on GPUs may not deliver expected performance as GPUs' relatively smaller cache can be easily congested by thousands of threads, causing serious contention and thrashing.

For example, the Intel Haswell CPU has 16 KB cache per thread per core available, but, NVIDIA Fermi GPU has only 32B cache per thread available, which is significantly smaller. Even if we consider the cache capacity per warp (i.e., a thread execution unit in GPU), it has only 1 KB per warp, which is still far smaller than the CPU cache. Generally, the per-thread or per-warp cache share for GPUs is much smaller than for CPUs. This suggests the useful data fetched by one warp is very likely to be evicted by other warps before actual reuse. Likewise, the useful data fetched

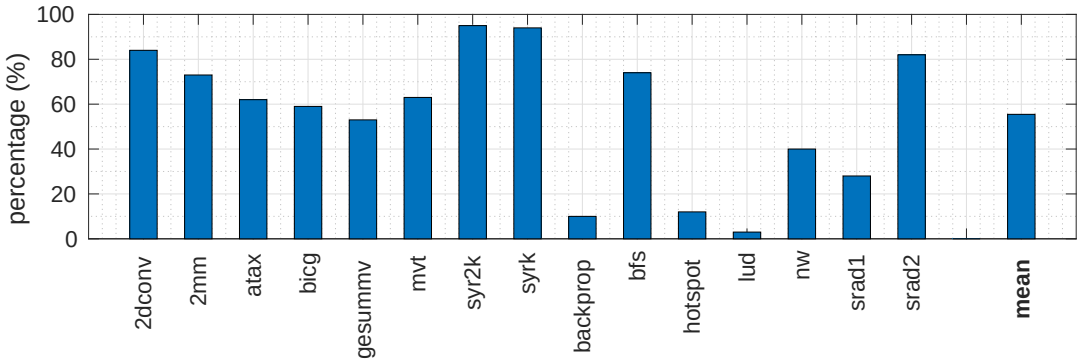


Figure 5.1. Stall time percentage over simulation cycle time.

by a thread can also be evicted by other threads in the same warp. Such eviction and thrashing conditions destroy discovered locality and impair performance. Moreover, the excessive incoming memory requests can lead to significant delay when threads are queuing for the limited resources in caches (e.g., a certain cache set, MSHR entries, miss buffers, etc).

This resource contention represents the cause of resource acquisition fail to service a request. The graph in Figure 3.5 shows which cache resource is a contention source and how much each resource contention occurs for each benchmark. The stall time over all simulation cycle time due to this resource contention is illustrated in Figure 5.1. Average stall cycle for all the benchmarks is about 55% of the total simulation cycle.

5.2.2 Low Cache Line Reuse

While GPGPU applications may exhibit good data reuse, due to the small size of the cache and heavy contention in L1D cache as well as many active memory requests, the distance between those accesses that could exploit reuse effectively become farther apart, resulting in a miss. As in Figure 3.7, the *Reuse0* dominates the distribution with about 85%. That is, many of the cache lines are polluted by the one time use data. If such data is detected before insertion and is avoided from insertion, the efficiency of the cache will increase.

5.3 Locality-Aware Selective Caching

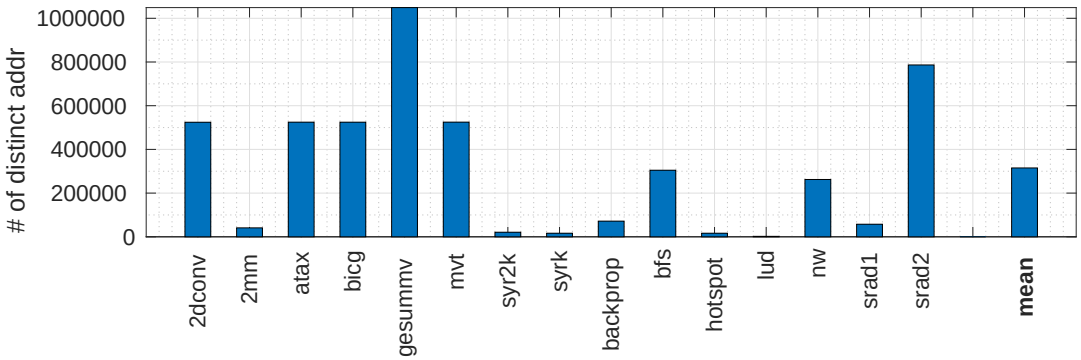
Cache line pollution by no-reuse memory requests causes severe cache contention. Caching lines likely to be reused and bypassing lines not likely to be reused is the key to locality-aware selective caching.

Existing CPU cache bypass techniques use memory addresses for bypass decision. The decision is based on hit rate of the memory access instructions [85], temporal locality [28], access frequency of the cache blocks [46], reuse distance [39], references and access intervals [50]. However, due to the massive parallel thread execution of GPUs, using memory addresses for bypass decision in GPUs is impractical. Figure 5.2a shows the number of distinct memory addresses present in the GPU benchmarks investigated. Hundreds of thousands of memory blocks are accessed during the execution of these kernels. On average, the number of distinct memory addresses is about 320,000.

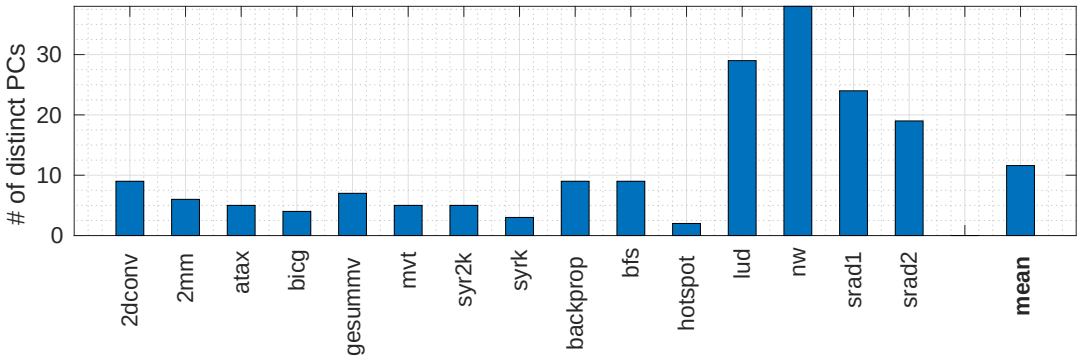
Compared to this, the number of load instructions in GPU kernels is relatively small. Due to the SIMD nature of the GPUs, the kernel code size is small and each thread shares the same code while executing with different data. Figure 5.2a presents that the number of distinct load instructions identified by their PC are small. The average number of distinct PCs is about 11 in the benchmarks tested. Therefore, keeping track of average reuse frequency per instruction indexed by PC appears to be a manageable solution for making the cache/bypass decision.

5.3.1 Reuse Frequency Table Design and Operation

We propose a *reuse frequency table* as in Figure 5.3 to store each load instruction's PC and reuse frequency. It has a *hashed-PC* field that stores the memory instruction's PC. As can be seen from Figure 5.2b, 64 entries would suffice to distinguish instructions. Because of the characteristic of hashing, a kernel with more than 64 instructions can still be accommodated by the table. The other field, *ReuseFrequencyValue*, holds the moving average of reuse frequency for



(a) The number of distinct memory addresses during the execution of kernels.



(b) The number of distinct PCs of the load instructions.

Figure 5.2. The number of addresses and instructions for load.

caching decisions. We revisit this entry in Section 5.3.2 for optimization.

This table is maintained globally to be shared by all the SMs since thread blocks are randomly distributed to each SM and their average memory access behavior is similar. Since this table is only updated when an eviction occurs in a cache, the frequency of update is lower than the frequency of cache accesses. Therefore, contention to update entries on this global structure is manageable.

Figure 5.4 shows the algorithm procedure in detail. It consists of two phases: reuse frequency table update and a cache/bypass decision as in Figure 5.4a and Figure 5.4b, respectively.

Reuse frequency table update (Figure 5.4a): when memory access is requested, it probes cache for a hit or a miss (①). A miss requires a decision whether or not a line has to be evicted from the cache set (②). When no eviction is needed (④), the request simply allocates a line

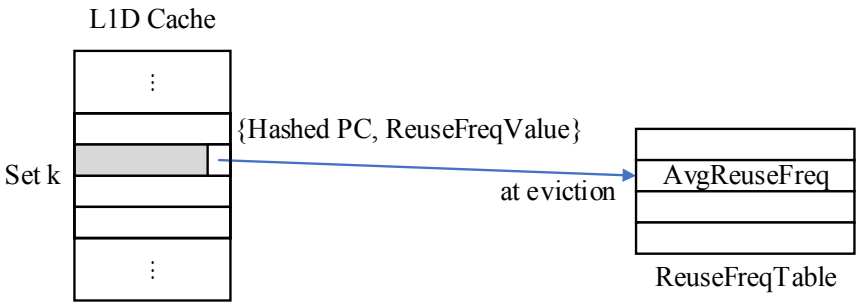
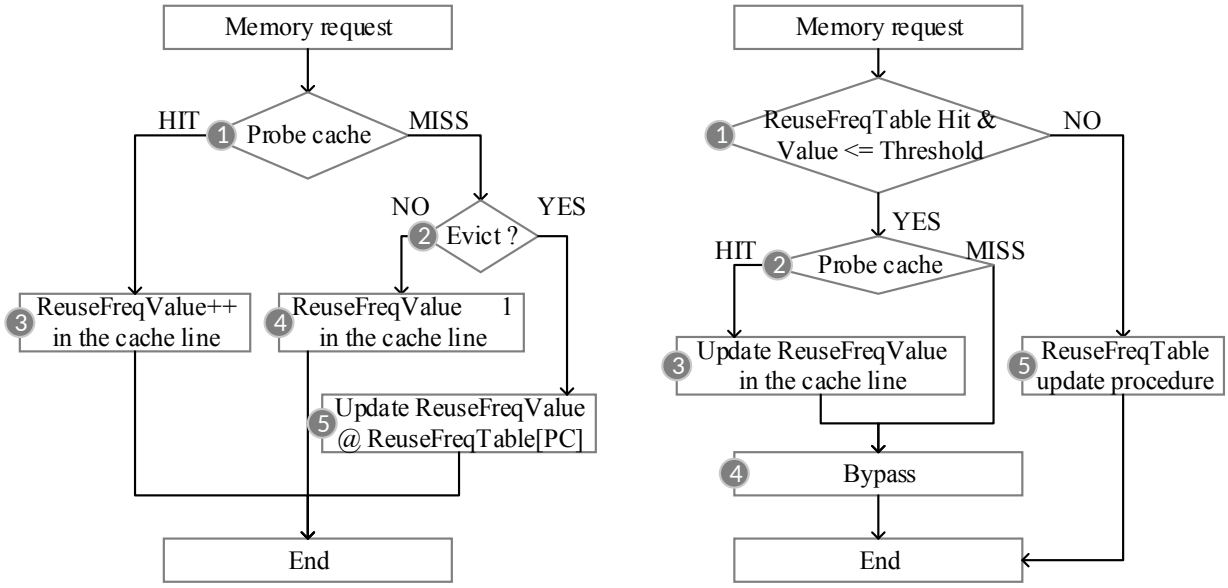


Figure 5.3. *Reuse frequency table* entry and operation.



(a) Reuse frequency table update procedure. (b) Caching decision.

Figure 5.4. Reuse frequency table update and caching decision.

with the hashed-PC and the ReuseFreqValue of 1. If an eviction is needed (⑤), the victim line's ReuseFreqValue updates the entry in the reuse frequency table indexed by the victim line's hashed-PC. If the request is a hit (③), the ReuseFreqValue in the cache line is increased by one.

In summary, the request updates *cache line* 1) when a request hits a cache line (reuse frequency increased by 1) and 2) when the request is a miss and no eviction occurs (reuse frequency set to 1). The request updates *reuse frequency table* with the evicted line's PC and ReuseFreqValue only when an eviction occurs in a cache.

Caching decision (Figure 5.4b): When memory access is requested, it probes the reuse

frequency table to see whether it is a hit or a miss. When it is a hit and the ReuseFreqValue in the entry is less than or equal to a threshold (①), it indicates the request is not likely to be referenced in the future. This request is determined to be bypassed. However, if the request is bypassed, there is no chance to update the reuse statistics for the PC. Then, the PC's reuse frequency becomes stale, and the LDST unit falsely bypasses memory requests of the PC. To avoid this situation, even when the LDST unit decides to bypass the memory request, it probes the cache (②) and if it is a hit, it updates the cache entry's ReuseFreqValue (③). The request is bypassed after that (④). When the memory request does not satisfy the bypass criterion, it follows the reuse frequency table update procedure (⑤).

5.3.2 Threshold Consideration

When a line is evicted from the cache, the ReuseFreqValue of the evicted line updates the AvgReuseFreq field in the reuse frequency table. The average value can be calculated either as a true average or a moving average. A true average calculation needs to track the number of evictions and the summation of all the reuse frequency values, while a moving average needs a previous average and a forgetting factor α. This calculation needs either value accumulation or floating point multiplication.

Figure 5.5 shows the IPC improvement over baseline with different thresholds. The result indicates that all the simulated threshold values improve performance. *Th1.0* shows the least improvement while *Th1.5* shows the most improved performance. However, when the threshold is greater than 1.0, which means some requests that have potential reuse are bypassed, some benchmarks such as `2dconv`, `2mm`, and `backprop` suffer performance degradation. To avoid such a degradation, we choose the threshold 1.0.

When a threshold value of 1.0 is used, we do not need to calculate an average of reuse frequency, but, we simply need to record the ReuseFreqValue 0 or 1, where 0 indicates *no-reuse*

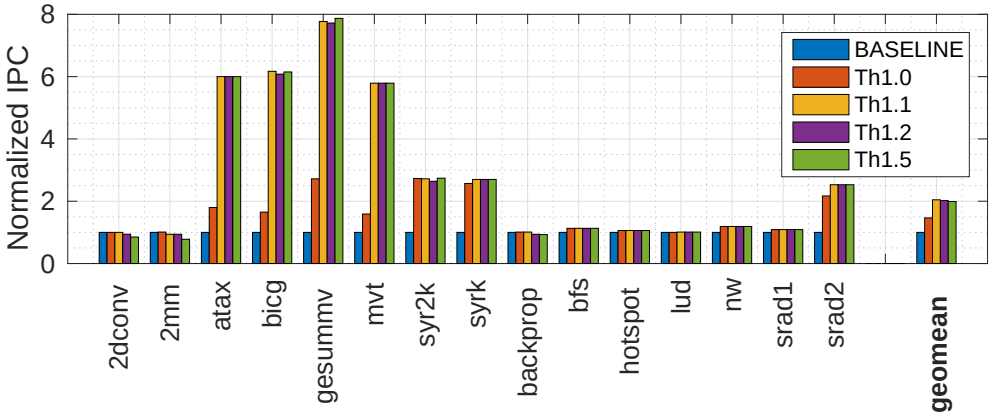


Figure 5.5. IPC improvement with different threshold values for caching decision.

and 1 indicates *reuse*. We now simplify the ReuseFreqValue field implementation in a cache line to 1 bit to hold the bypass-or-not information. Our reuse frequency table also can be simplified to have 1 bit for AvgReuseFreq.

5.3.3 Algorithm Features

1 bit for cache/bypass decision: From the threshold simulation analysis in Section 5.3.2, we minimize the implementation of ReuseFreqValue and AvgReuseFreq to 1 bit. The initial load to a cache line sets the bit to 0, indicating *no-reuse*. Whenever reuse occurs, it flips the bit to 1, indicating *reuse* in the cache line. When the Reuse Frequency Table is updated upon cache line eviction, the current value in the table will be ORed with the new value. This significantly simplifies the design complexity and latency. We do not need to hold several bits for the reuse frequency value in a cache line nor to calculate a running average for the table entry.

Conservative bypassing: The selective caching decision using the reuse frequency value for our proposed scheme is conservative. When memory requests from the same instruction (PC) have a different reuse characteristic, that is, some requests are *no-reuse* and the others are *reuse*, then the reuse frequency in the reuse frequency table is set to *reuse*. Then, the memory requests from the PC are determined not to be bypassed. Likewise, when multiple PCs are mapped to the

same hashed-PC entry, only when both of the PCs' requests are *no-reuse*, the requests from those PCs are bypassed. Otherwise, the requests are not bypassed. This guarantees no performance degradation when the threshold value is 1.0.

Avoid stale ReuseFreqValue statistics: In Figure 5.4b, ③ updates the ReuseFreqValue in the cache line even if the request is to be bypassed. Without the step ③, the ReuseFreqValue would become stale once a bypass decision is made for a PC. Step ③ keeps updating the ReuseFreqValue whenever a hit occurs. This changes the status of the PC from *no-reuse* to *reuse* so that the request with the PC is not bypassed the next time.

5.3.4 Contention-Aware Selective Caching Option

While locality-aware selective caching increases IPC performance by reducing cache pollution, there is still another cause for a large portion of the resource contention. The previously mentioned approach, *Contention-aware Selective Caching* in Chapter 4 addresses one source of a large portion of the resource contention as coincident intra-warp contention which is mainly caused by a column-strided pattern. The chapter identifies that the coincident intra-warp contention severely blocks the overall cache resources. The selective caching detects the problematic memory divergence, identifies the congested sets, and decides caching or not. Since this locality-aware selective caching can be used along with contention-aware selective caching without interfering with each other, a synergistic effect is expected.

5.4 Experiment Methodology

5.4.1 Simulation Setup

We configured and modified GPGPU-Sim v3.2 [82], a cycle-accurate GPU architecture simulator, to find contention in the GPU memory hierarchy, and implemented the proposed algorithms. The NVIDIA GTX480 hardware configuration is used for the system description. The

Number of SMs	15
SM configuration	1400 Mhz, SIMD Width: 16 Warp size: 32 threads Max threads per SM: 1536 Max Warps per SM: 48 Warps Max Blocks per SM: 8 blocks
Warp schedulers per core	2
Warp scheduler	Greedy-Then-Oldest (GTO) [73] (default) Loose Round-Robin (LRR) Two-Level (TL)[63]
Cache/SM	L1 Data: 16 KB 128B-line/4-way (default) L1 Data: 16 KB 128B-line/2-way L1 Data: 16 KB 128B-line/8-way Rep. Policy: Least Recently Used (LRU) Shared memory: 48 KB
L2 unified cache	768 KB, 128 B line, 16-way
Memory partitions	6
Instruction dispatch throughput per scheduler	2 instructions per cycle
Memory Scheduler	Out of Order (FR-FCFS)
DRAM memory timing	t_{CL}=12, t_{RP}=12, t_{RC}=40, t_{RAS}=28, t_{RCD}=12, t_{RRD}=6
DRAM bus width	384 bits

Table 5.1. Baseline GPGPU-Sim configuration.

baseline GPGPU-Sim configurations for this chapter are summarized in Table 5.1.

5.4.2 Benchmarks

To perform our evaluations, we chose benchmarks from Rodinia [14] and PolyBench/GPU [31]. We pruned our workload list by omitting the applications provided in the benchmark for basic hardware profiling and graphics interoperability. We also omitted kernels that did not have a grid size large enough to fill all the cores and whose grid size could not be increased without significantly changing the application code. The benchmarks tested are listed in Table 5.2.

Name	Suite	Description
2DCONV	PolyBench	2D Convolution
2MM	PolyBench	2 Matrix Multiplication
ATAX	PolyBench	Matrix Transpose and Vector Mult.
BICG	PolyBench	BiCG SubKernel of BiCGStab Linear Sol.
GESUMMV	PolyBench	Scalar, Vector and Matrix Multiplication
MVT	PolyBench	Matrix Vector Product and Transpose
SYR2K	PolyBench	Symmetric rank-2k operations
SYRK	PolyBench	Symmetric rank-k operations
BACKPROP	Rodinia	Back propagation
BFS	Rodinia	Breadth-First Search
HOTSPOT	Rodinia	Hot spot physics simulation
LUD	Rodinia	LU Decomposition
NW	Rodinia	Needleman-Wunsch
SRAD1	Rodinia	Speckle Reducing Anisotropic Diffusion
SRAD2	Rodinia	SRAD 2D

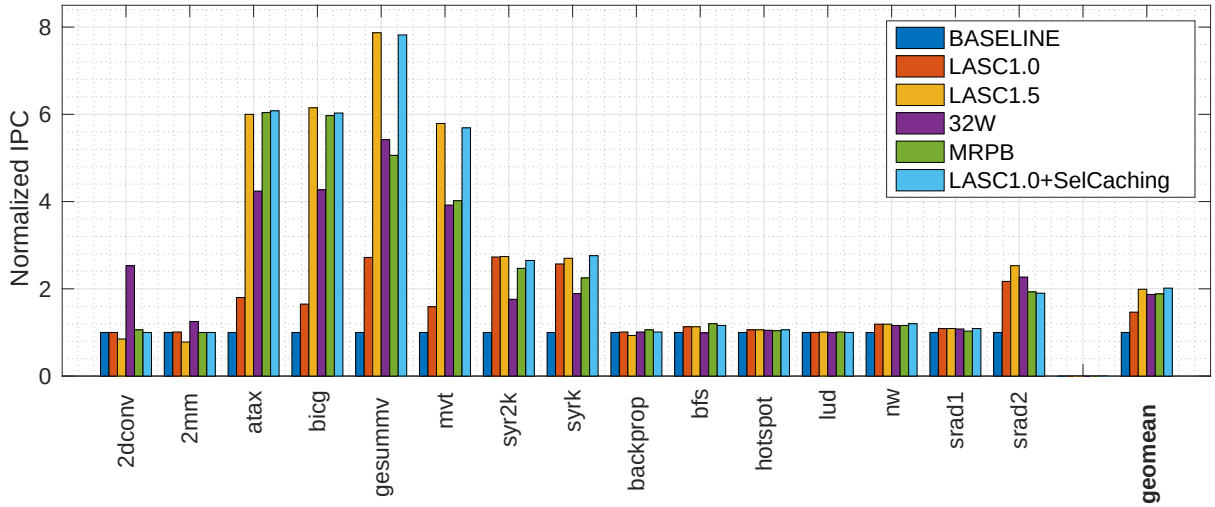
Table 5.2. Benchmarks from PolyBench [31] and Rodinia [14].

5.5 Experimental Results

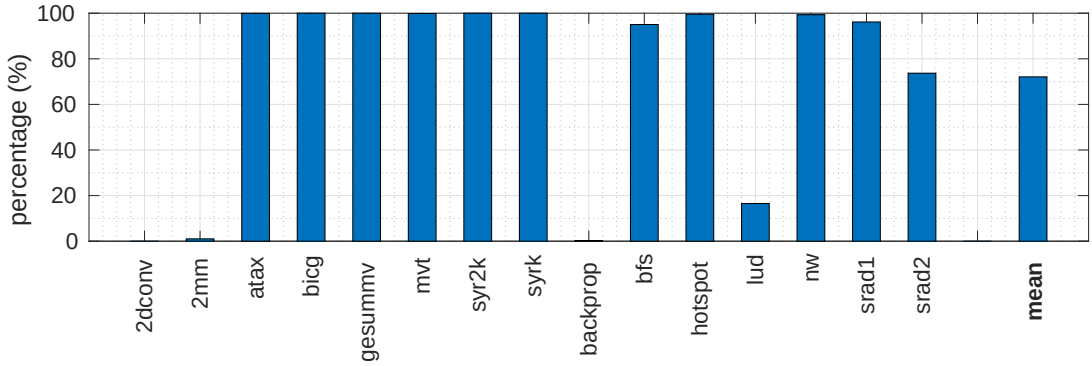
5.5.1 Performance Improvement

Figure 5.6a shows the normalized IPC improvement of our proposed algorithm over the baseline. The baseline graph is shown with a normalized IPC of one. *LASC1.0* and *LASC1.5* are the proposed schemes with different threshold values for comparison. *LASC1.0+SelCaching* represents the scheme with contention-aware selective caching. For performance comparison with other state-of-the-art techniques using bypassing, we choose a memory request prioritization, *MRPB* [42].

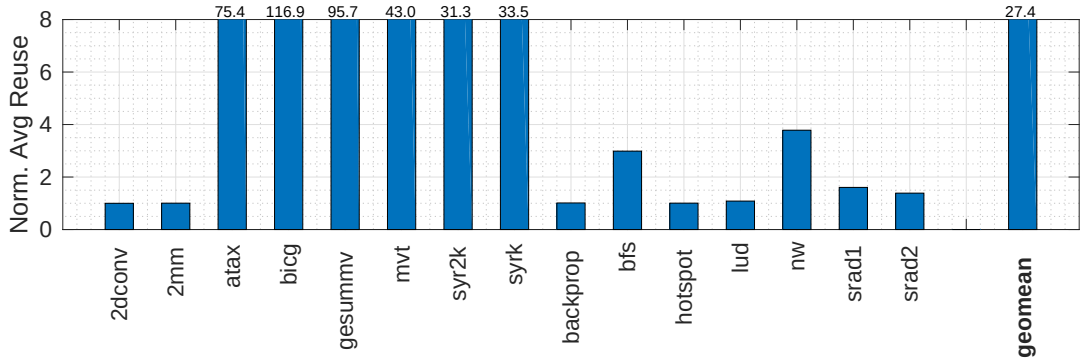
Note that there is no performance degradation with threshold value 1.0. When the threshold value is greater than 1.0, benchmarks suffer from the performance degradation due to excessive bypassing. LASC1.0 performance enhancement over baseline is about 1.39x. Our proposed scheme with contention-aware selective caching described in Chapter 4, *LASC1.0+SelCaching*, outper-



(a) IPC improvement.



(b) Reduction of no-reuse memory request in percentage.



(c) Reuse Frequency enhancement.

Figure 5.6. Overall improvement: IPC improvement and L1D cache access reduction.

forms baseline by 2.01x and *MRPB* by 7%.

Figure 5.6b shows the reduction of no-reuse memory requests. For most of the benchmarks, `atax, bicg, gesummv, mvt, syr2k, syrk, bfs, hotspot, nw`, and `srad1`, more than 99% of the no-reuse memory requests are bypassed and do not pollute the cache. On average, 73% of the no-reuse memory requests are bypassed. The reason for less removal in some benchmarks such as `2dconv, 2mm, backprop, lud`, and `srad2` is because the no-reuse requests are mixed with reuse requests within the same PCs. As described in Section 5.3.3, since our scheme is conservatively bypassing, those requests are not bypassed.

Figure 5.6c shows the enhanced reuse frequency when *LASC1.0+SelCaching* is used. With the average reuse frequency for the baseline one, 27x more references are made in the cache on average. Notably, for the `bicg` benchmark, the average reuse frequency is enhanced by 117x. Since most of the no-reuse traffic has been bypassed, there are more chances for other requests to be referenced.

5.5.2 Effect of Warp Scheduler

Our experiment used the *Greedy-Then-Oldest (GTO)* warp scheduler as a default warp scheduler. Since our locality-aware selective caching algorithm improves performance based on the reuse frequency, other schedulers should not affect the trend of overall performance improvement. Figure 5.7 shows the results with different warp schedulers, such as the *TwoLevel* [63] and *RoundRobin* warp scheduler. Since GTO is designed to reduce cross-warp contention and TwoLevel is designed to improve core utilization by considering branch diversity, GTO scheduler enhancement is the least while the RR scheduler enhancement is the most. Overall, the experiments demonstrate that the locality-aware selective caching scheme improves the performance effectively regardless of scheduler.

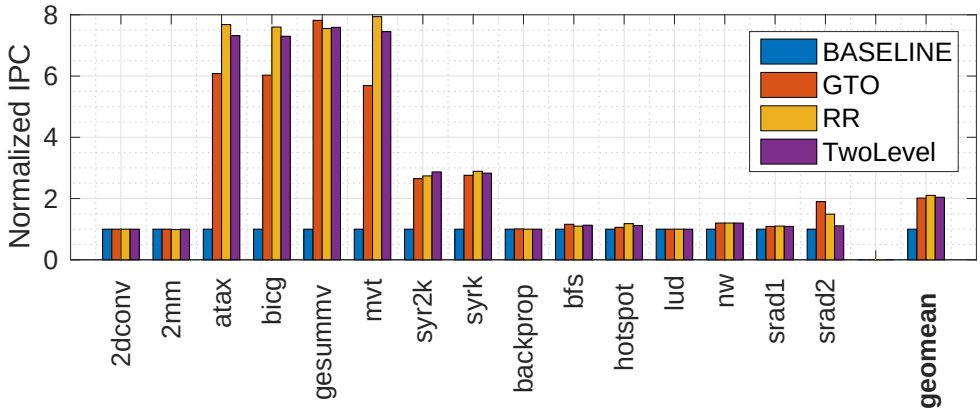


Figure 5.7. IPC improvement with different schedulers.

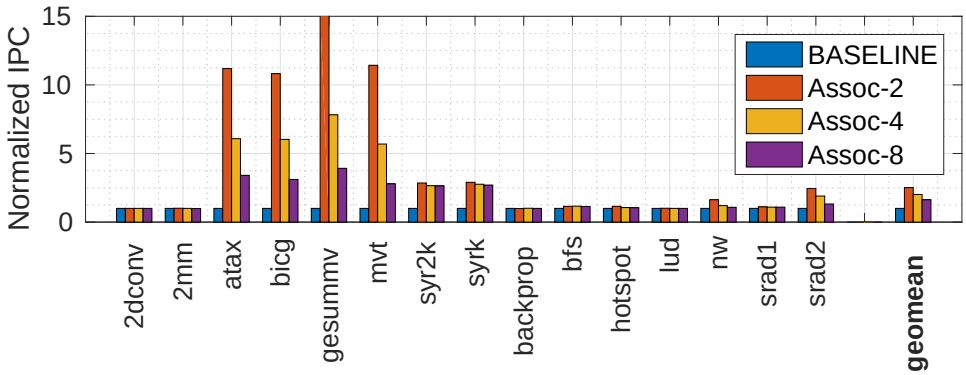


Figure 5.8. IPC improvement with different associativities.

5.5.3 Effect of Cache Associativity

Figure 5.8 shows the results with different cache associativities: 2-, 4-, and 8-way. Cache contention tends to be much more severe with smaller associativity cache. Generally, memory requests become more congested in smaller associativity and the stall becomes worse compared to larger associativity cases. Therefore, the IPC improvement for associativity 2 is the most among the three different associativity cases. Cache with associativity 8 still improves the performance about 1.63x.

5.6 Related Work

5.6.1 CPU Cache Bypassing

Much of the existing research focuses on CPU cache management techniques [28, 38, 39, 46, 71, 85, 87]. We only show bypassing related techniques here. Among these, a selection of papers have explored bypassing in CPU caches. Tyson et al. proposed bypassing based on the hit rate of the memory access instructions [85], while Johnson et al. propose to use the access frequency of the cache blocks to predict bypassing [71]. Kharbutli and Solihin propose using counters of events such as number of references and access intervals to make bypass predictions in the CPU last-level cache [50]. All of these techniques use memory address-related information to make the prediction, costing significant storage overhead that would be impractical for GPU caches. Program counter trace-based dead block prediction [53] leveraged the fact that sequences of memory instruction PCs tend to lead to the same behavior for different memory blocks. This dead block prediction scheme is useful for making bypass predictions in CPUs. We show that GPU kernels are small with few distinct memory instructions. Using only the PC of the last memory instruction to access a block is sufficient for a GPU bypassing prediction. Cache Bursts [59] is another dead block prediction technique that exploits bursts of accesses hitting the MRU position to improve predictor efficiency. For GPU workloads that use scratch-pad memories, the majority of re-references have been filtered. Gaur et al. [25] proposed bypass and insertion algorithms for exclusive LLCs to adaptively avoid unmodified dead blocks being written into the exclusive LLC.

5.6.2 GPU Cache Bypassing

Jia et al. [41] justified the effect of the cache in GPUs. They presented a simulation result that L1D cache may degrade the overall system performance. Then, they proposed a static method to analyze GPU programs and determine if caching is beneficial or detrimental at compile time and to apply it to control the L1D cache to bypass or not. Jia et al. [42] later proposed a memory

request prioritization buffer (MRPB) to improve GPU performance. MRPB prioritized the memory requests in order to reduce the reuse distance within a warp. It also used cache bypassing to mitigate intra-warp contention. When bypassing, it blindly bypassed memory requests whenever they detect resource contention. Therefore, there are some benchmarks that suffer from performance degradation. Compared to MRPB, our locality-aware selective caching does not degrade performance since we measure the reuse frequency dynamically and conservatively decide caching according to the reuse frequency.

Rogers et al. proposed cache-conscious wavefront scheduling (CCWS) to improve GPU cache efficiency by avoiding data thrashing that causes cache pollution [72]. CCWS estimates working set size of active warps and dynamically restricts the number of warps. This may adversely affect the ability to hide high memory access latency of GPUs. Our selective caching bypasses the no-reuse blocks without under-utilizing the SIMD pipeline to reduce cache thrashing.

Lee and Kim proposed a thread-level-parallelism-aware cache management policy to improve performance of the shared last level cache (LLC) in heterogeneous multi-core architecture [55]. They focus on shared LLCs that are dynamically partitioned between CPUs and GPUs. Mekkat et al. proposed a similar idea for heterogeneous LLC management [60], to better partition LLC for GPUs and CPUs in a heterogeneous system.

Li et al. [57] exploits bypassing to reduce contention. They extract locality information during compile time and throttle the warp scheduler to avoid thrashing due to warp contentions. Their work focuses on reducing cross-warp contention. Static analysis does not reflect the dynamic behavior of the application.

Tian et al. [80] also exploits PC to predict bypassing. Their method maintains a table for bypass prediction. They use confidence count to control bypassing. Every reuse decreases the confidence count and every miss increases the confidence count. When the confidence count is greater than a predetermined value for the PC of a memory request, then the request is bypassed. How-

ever, this scheme takes a training time until it actually bypasses a request since it uses a confidence counter and the tables are maintained for each L1D cache. To compensate for misprediction, they use a bypassBit in the L2 cache. However, when the bit is set and reset by multiple SMs' requests, its bypass decision is not accurate. Our scheme maintains a global reuse frequency table to reflect the program's overall behavior. Also, our scheme dynamically updates the bypass table, even by bypassed requests, to avoid misprediction.

5.7 Summary

This chapter shows that the massive parallel thread execution of GPUs can result in memory access patterns that cause significant cache resource contention. By observing the resource contention classification and reuse statistics, we also identify that the streaming requests are dominant in the memory requests and they severely pollute the cache, resulting in severe degradation on the overall performance.

We proposed a locality-aware selective caching algorithm based on per-PC memory reuse frequency in the L1 data (L1D) cache on GPUs. When we choose 1.0 for the caching threshold, the design becomes very simple to deploy. Our scheme uses a conservative bypassing approach, ensuring that no performance degradation occurs.

Our proposed scheme improves the IPC performance by 1.39x over the baseline. Together with contention-aware selective caching, it improves the overall performance by 2.01x. Our scheme outperforms the state-of-the-art bypassing mechanism, MRPB, by 7%. Our algorithm also reduces 73% of the no-reuse memory requests helping to avoid polluting the L1D cache and enhances the average reuse frequency by 27x.

CHAPTER 6

MEMORY REQUEST SCHEDULING

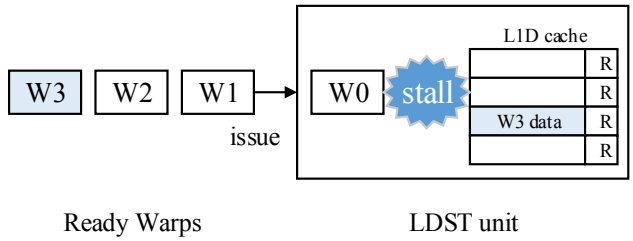
6.1 Introduction

Column-strided memory access patterns are a source of cache contention commonly found in data-intensive applications. The access pattern is compiled to warp instructions and generates memory access patterns that are not well coalesced, causing resource contention. The resulting memory accesses are serially processed in the LDST unit, stress the L1 data (L1D) caches, and result in serious performance degradation.

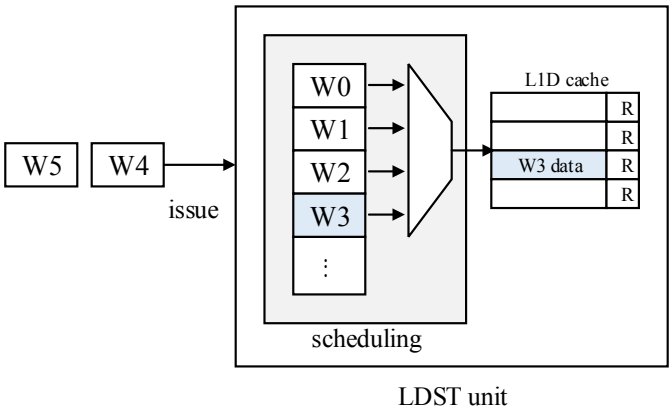
Cache contention also arises from cache pollution caused by low reuse frequency data. For the cache to be effective, a cached line must be reused before its eviction. But, the streaming characteristic of GPGPU workloads and the massively parallel GPU execution model increase the effective reuse distance, or equivalently reduce reuse frequency of data. In GPUs, the pollution caused by a low reuse frequency (i.e., large reuse distance) data is significant.

Due to the contention, the LDST unit becomes saturated and stalled. A stalled LDST unit does not execute memory requests from any ready warps in the previous issue stage. The warp in the LDST unit retries until the cache resource becomes available. During this stall, the private L1D cache is also in a stall, so no other warp requests can probe the cache. However, there may be data in an L1D cache which may hit if other ready warps in the issue stage could access the cache. In such cases, the current structure of the issue stage and L1D unit execution do not allow the provisional cache probing. This stall prevents the potential hit chances for the ready warps.

This chapter proposes a memory request schedule queue that holds ready warps' memory



(a) LDST unit is in a stall. A memory request from ready warps cannot progress because the previous request is in stall in the LDST unit.



(b) LDST unit with scheduling queue.

Figure 6.1. LDST unit in stall and the scheduling queue.

requests and a scheduler that effectively schedules them in a manner that increases the chances of a hit. In this chapter, we identify the problematic GPU LDST unit stalls and analyze the potential hit in depth. Then, we detail the various design factors of a queue mechanism and scheduling policy. A thorough analysis of the experimental results follows.

6.2 Cache Contention

6.2.1 Memory Request Stall due to Cache Resource Contention

When the LDST unit becomes saturated, a new request to the LDST unit will not be accepted. When the LDST unit is stalled by one of the cache resources, for example, a line, an MSHR entry or a miss queue entry, the request fully owns the LDST unit and retries for the resource. Whenever the contending resource becomes free, the retried request finally acquires the

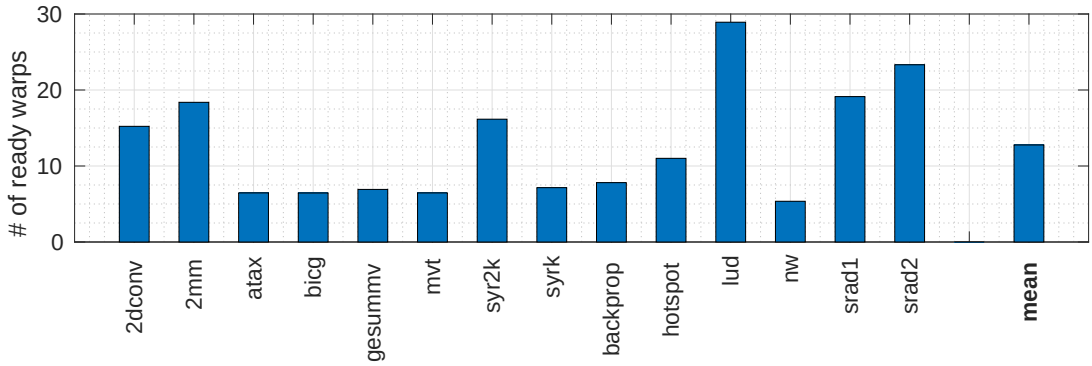


Figure 6.2. The average number of ready warps when cache resource contention occurs.

resource and the LDST unit can accept the next ready warp. While the stalled request retries, the LDST unit is blocked by the request and cannot be preempted by another request from other ready warps. Usually, the retry time is large because the active requests occupy the resource during the long memory latency to fetch data from the lower level of the cache or the global memory.

During this stall, other ready warps which may have the data they need in the cache cannot progress because the LDST unit is stalled by the previous request. This situation is illustrated in the Figure 6.1a. Assume the $W0$ is stalled in the LDST unit and there are 3 ready warps in the issue stage. While $W0$ is stalled, the 3 ready warps in the issue stage cannot issue any request to the LDST unit. Even when the $W3$ data is in the cache at that moment, it cannot be hit by $W3$. While $W0$, $W1$, and $W2$ are being serviced in the LDST unit, the $W3$ data in the cache may be evicted by the other requests from $W0$, $W1$, or $W2$. If that happens what would have been a $W3$ hit now misses in the L1D cache and must send a fetch request to the lower level to recall the data. If $W3$ had a chance to be scheduled to the stalled LDST unit, it would hit and save the extra cycles to fetch the data from the lower level.

To estimate the potential opportunity for a hit under these circumstances, we measured the number of ready warps when a memory request stall occurs. This number does not directly represent the increase in the number of hits, but it gives us a potential estimate. From Figure 6.2, about 12 warps on average are ready to be issued when the LDST unit is in a stall.

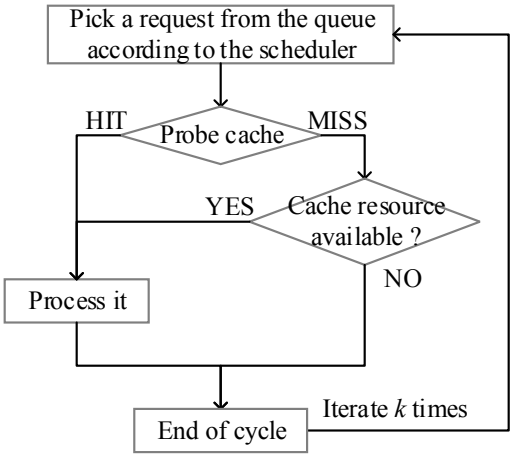


Figure 6.3. A procedure for the memory request scheduler.

6.3 Memory Request Scheduling

6.3.1 Memory Request Queuing and Scheduling

To enable potential hits under LDST unit stall, we introduce a memory request queue for rescheduling the memory request upon stall.

Queuing: When the issue unit issues a warp instruction, instead of monitoring the status of the LDST unit, the memory requests from the instruction are pushed into the appropriate slot depending on the warp ID. When the queue is full for the slot, then the warp backs up and retries at the next cycle since the LDST unit has memory requests to schedule from the same warp.

Scheduling: In the LDST unit, one request from the queue is picked by a scheduler and it probes the for a hit. If it hit, the instruction is executed in the LDST unit. When it is a miss and the resource is available to process the instruction, then the instruction is executed in the LDST unit. However, when the resource is not available, the next available warp instruction is picked by a scheduler and repeats the process depending on an iteration factor, k. This iteration factor, k, is implemented as multiple cache probe units to be probed concurrently.

Our design allows the memory request queue to hold up to n number of requests generated by a warp instruction per slot. The queue size is the same as the maximum number of warps per

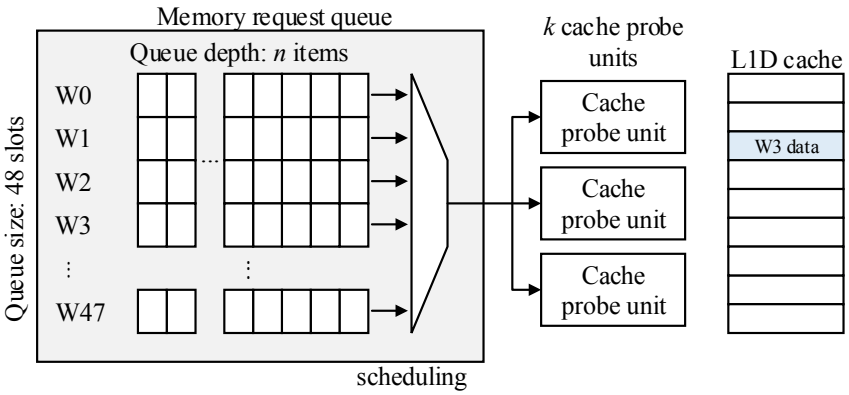


Figure 6.4. A detailed view of the memory request queue and scheduler.

SM, which is 48 in our simulation setup. This scheme is illustrated in the Figure 6.1b and the flow is in Figure 6.3.

6.3.2 Queue Depth

When the issue unit issues a warp instruction, the generated memory requests are pushed into the memory request queue. Since one warp instruction can generate up to the SIMD lane size of memory access requests, the depth of the queue for a warp must be at least 32 slots to fully queue the generated memory request. If the depth is smaller than 32, the LDST unit is still in a stall. The queue depth is illustrated in Figure 6.4.

6.3.3 Scheduling Policy

After memory requests are enqueued to an appropriate queue slot, a scheduling policy imposes a service priority between these requests. One straightforward scheduling policy is to apply a *fixed-order policy (FIXED)*, i.e., an item from queue i can be scheduled only if queues 1 to $i-1$ have been emptied. Another policy is a *round-robin policy (RR)* which selects items from queues in turn, i.e., one item per non-empty queue. Another policy is a *grouped round-robin policy (GRR)* which allows multiple items from consecutive slots (one item per slot) in the queue to probe cache at the same time. The factor, k, depends on the hardware support. For example, with 4-port

cache support, four probes can be done at the same time.

The multiple cache probe units are illustrated in Figure 6.4. k items are chosen in a round-robin fashion to probe cache for potential hits. When multiple hits occur in a group, the earlier slot in a group is serviced in that cycle, and the next ks are probed in the next cycle. The multiple units require multiple L1D cache ports, but increasing the number of probe units reduces contention and increases the probability of a hit.

6.3.4 Contention-Aware Selective Caching Option

While memory request scheduling increases the hit rate when the LDST unit stall due to the resource contention, there are other resource contention causes. Chapter 4 addresses resource contention arising from *coincident intra-warp contention*, which is mainly caused by column-strided pattern. The chapter shows that the coincident intra-warp contention severely blocks the overall cache resources. It proposes selective caching which detects the problematic memory divergence, identifies the congested sets, and decides to cache or not using per-PC reuse count information. Since the proposed memory request scheduling can be used along with selective caching without interfering with each other, selective caching can increase cache hits.

6.4 Experiment Methodology

6.4.1 Simulation Setup

We configured and modified GPGPU-Sim v3.2 [82], a cycle-accurate GPU architecture simulator to find miss contention and resource contention in current GPU memory hierarchy, and implemented the proposed algorithms. The NVIDIA GTX480 hardware configuration is used for the system description. The baseline GPGPU-Sim configurations for this study are summarized in Table 6.1.

Number of SMs	15
SM configuration	1400 Mhz, SIMD Width: 16 Warp size: 32 threads Max threads per SM: 1536 Max Warps per SM: 48 Warps Max Blocks per SM: 8 blocks
Warp schedulers per core	2
Warp scheduler	Greedy-Then-Oldest (GTO) [73] (default) Loose Round-Robin (LRR) Two-Level (TL)[63]
Cache/SM	L1 Data: 16 KB 128B-line/4-way (default) L1 Data: 16 KB 128B-line/2-way L1 Data: 16 KB 128B-line/8-way Rep. Policy: Least Recently Used (LRU) Shared memory: 48 KB
L2 unified cache	768 KB, 128 B line, 16-way
Memory partitions	6
Instruction dispatch throughput per scheduler	2 instructions per cycle
Memory Scheduler	Out of Order (FR-FCFS)
DRAM memory timing	t_{CL}=12, t_{RP}=12, t_{RC}=40, t_{RAS}=28, t_{RCD}=12, t_{RRD}=6
DRAM bus width	384 bits

Table 6.1. Baseline GPGPU-Sim configuration.

6.4.2 Benchmarks

To perform our evaluations, we chose benchmarks from Rodinia [14] and PolyBench/GPU [31]. We pruned our workload list by omitting the applications provided in the benchmark for basic hardware profiling and graphics interoperability. We also omitted kernels that did not have a grid size large enough to fill all the cores and whose grid size could not be increased without significantly changing the application code. The benchmarks tested are listed in Table 6.2.

Name	Suite	Description
2DCONV	PolyBench	2D Convolution
2MM	PolyBench	2 Matrix Multiplication
ATAX	PolyBench	Matrix Transpose and Vector Mult.
BICG	PolyBench	BiCG SubKernel of BiCGStab Linear Sol.
GESUMMV	PolyBench	Scalar, Vector and Matrix Multiplication
MVT	PolyBench	Matrix Vector Product and Transpose
SYR2K	PolyBench	Symmetric rank-2k operations
SYRK	PolyBench	Symmetric rank-k operations
BACKPROP	Rodinia	Back propagation
HOTSPOT	Rodinia	Hot spot physics simulation
LUD	Rodinia	LU Decomposition
NW	Rodinia	Needleman-Wunsch
SRAD1	Rodinia	Speckle Reducing Anisotropic Diffusion
SRAD2	Rodinia	SRAD 2D

Table 6.2. Benchmarks from PolyBench [31] and Rodinia [14].

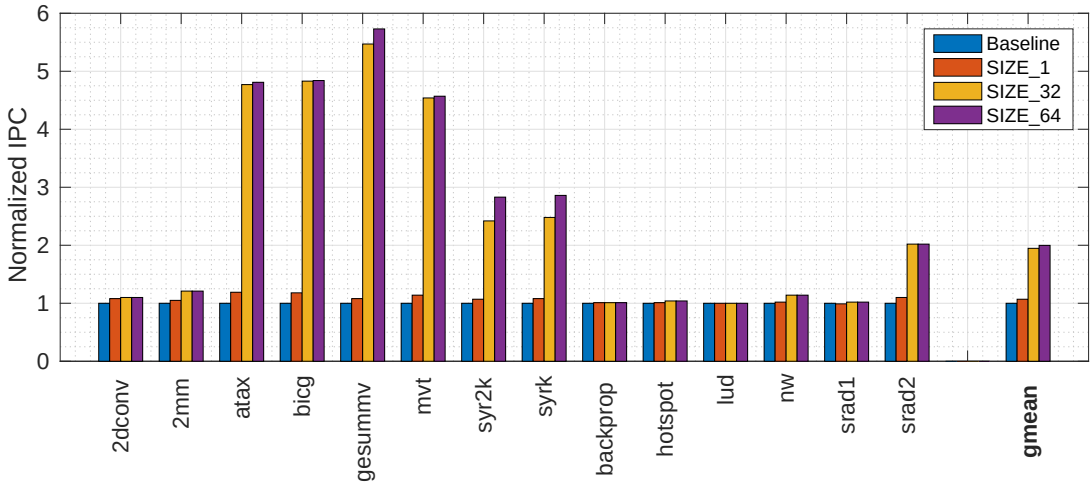
6.5 Experimental Results

6.5.1 Design Evaluation

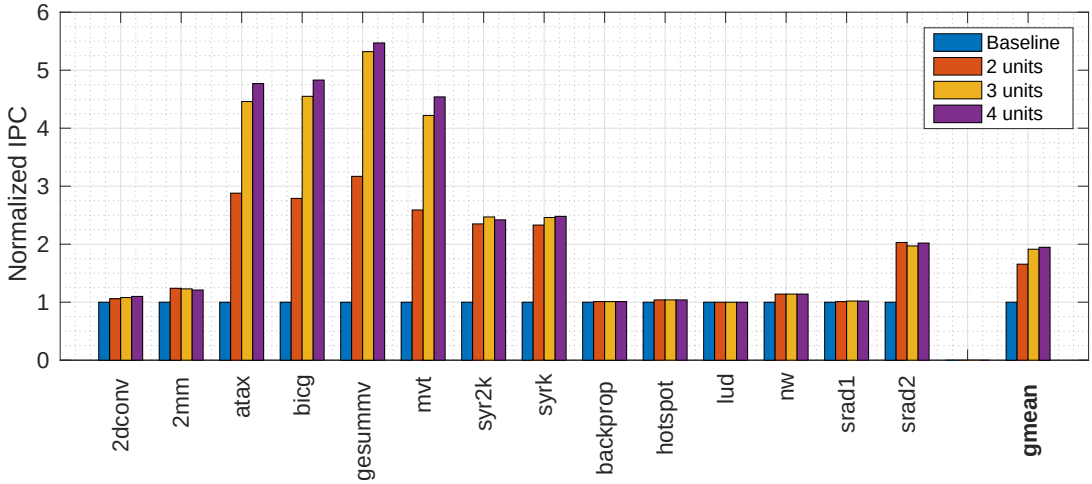
Figure 6.5 shows the normalized IPC improvement using different design implementations.

Queue Depth: Figure 6.5a shows the result using different queue depths such as 1, 32 and 64. When the queue depth is 1, the warp scheduler cannot issue all the generated memory access requests for the warp when more than 1 memory request is generated. Therefore, the scheduler still stalls on that instruction. When the queue depth is 32 or larger, the queue can hold all generated memory access requests and the warp scheduler is free to issue the next warp instruction. Figure 6.5a shows that the queue sizes 32 and 64 give similar results since the queue size 32 is large enough to free the warp scheduler.

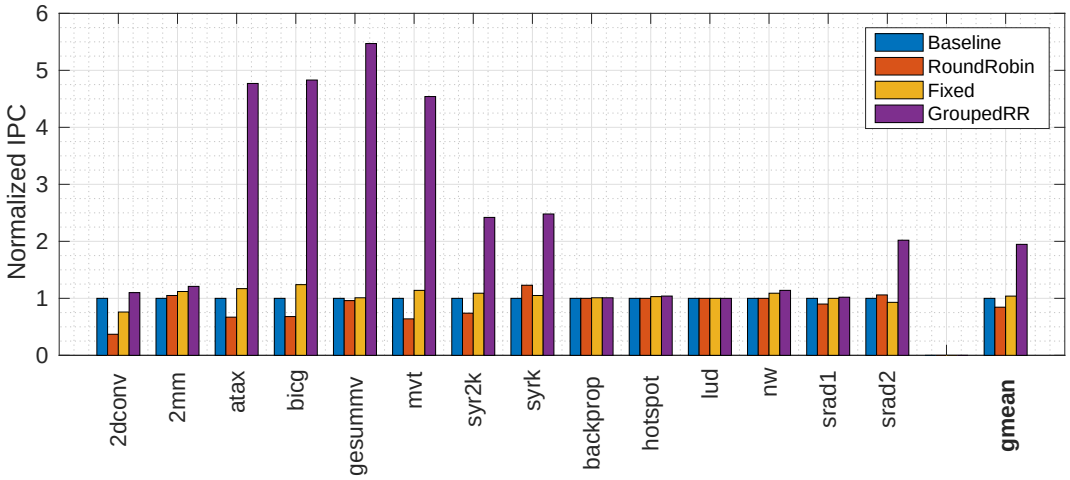
Multiple Cache Probe Units: Multiple cache probe units add extra hardware complexity, but saves execution cycles by improving the probability of a hit. The results with different factors such as 2, 3, and 4 are shown in Figure 6.5b. As expected, more probe units give better perfor-



(a) IPC improvement with different queue depths.



(b) IPC improvement with different cache probe unit counts.



(c) IPC improvement with different queue schedulers.

Figure 6.5. IPC improvement with different implementations.

mance. The factor of 4 is chosen as our design choice.

Memory Request Scheduling Policy: We experimented with multiple scheduling policies. Figure 6.5c shows the three different schedulers studied: *Fixed*, *RoundRobin*, and *GroupedRR*. *Fixed* scheduler schedules an item from queue i only if queues 1 to $i-1$ have been emptied. *RoundRobin* scheduler selects an item from the queue in turn, one item per non-empty queue. *GroupedRR* picks items from queue i to $i-k-1$ with the factor k. With the factor of four, four cache probe units are implied. As expected, *GroupedRR* outperforms other schedulers with the help of multiple probing units.

6.5.2 Performance Improvement

From the design evaluation in Section 6.5.1, we choose the queue depth of 32, 4 cache probe units, and the GroupedRR scheduling policy. Figure 6.6 shows the normalized IPC improvement of our proposed algorithm over the baseline. The baseline graph is shown with a normalized IPC of 1. *MemSched* represents the proposed memory request scheduling. *MemSched+SelCaching* represents the combined scheme with the proposed memory request scheduling and the selective caching as in Section 6.3.4. For performance comparison with another technique, we choose memory request prioritization, *MRPB* [42].

MemSched performance enhancement over baseline is about 1.95x. *MemSched* without any bypassing scheme has similar performance to the state-of-the-art bypassing scheme *MRPB*. The proposed scheme with the contention-aware selective caching described in Chapter 4, *MemSched+SelCaching*, outperforms baseline by 2.06x and the *MRPB* by 7%.

6.5.3 Effect of Warp Scheduler

Our experiment uses the *Greedy-Then-Oldest (GTO)* warp scheduler as a default warp scheduler. Since our memory request scheduling algorithm improves performance by finding potential hits during a LDST unit stall, other warp schedulers should not affect the trend of overall

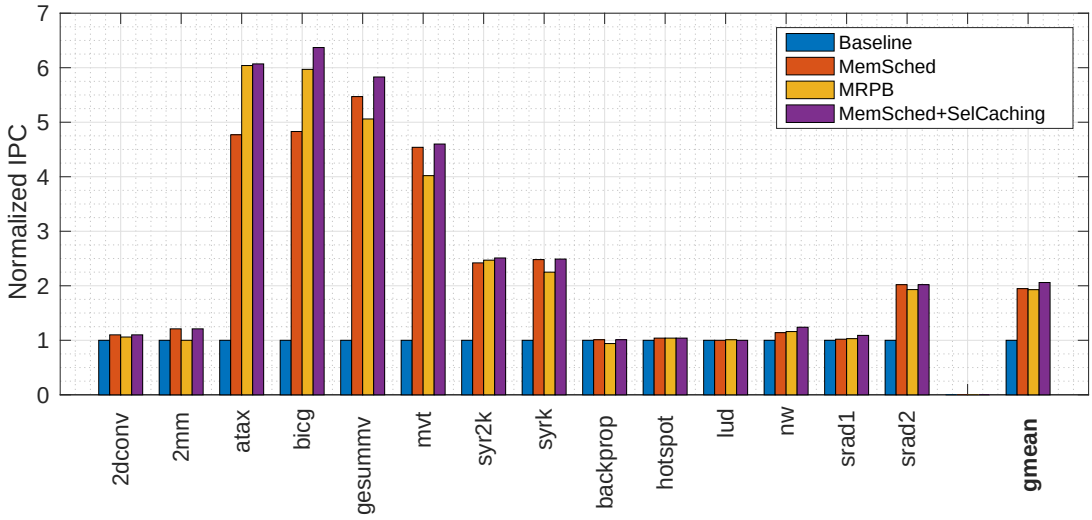


Figure 6.6. Overall IPC improvement.

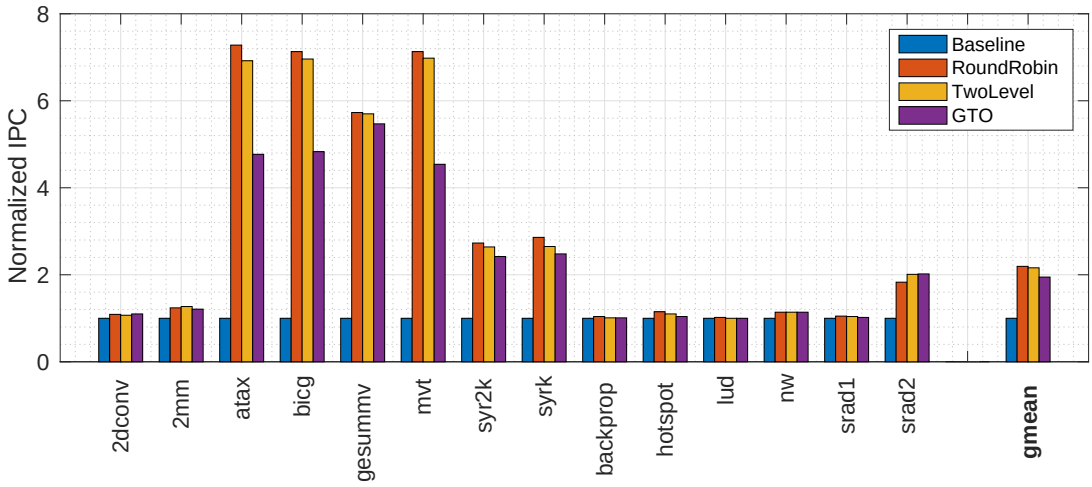


Figure 6.7. IPC improvement with different schedulers.

performance improvement. Figure 6.7 shows the results with different warp schedulers, such as the *TwoLevel* [63] and *RoundRobin* warp scheduler. Overall, experiments demonstrate that the memory request scheduling scheme improves the performance effectively regardless of scheduler.

6.5.4 Effect of Cache Associativity

Figure 6.8 shows the result with different cache associativities: 2-, 4-, and 8-way. Cache contention tends to be much more severe with a smaller associativity cache. Generally, memory requests in smaller associativity become more congested and the stall becomes worse than in other

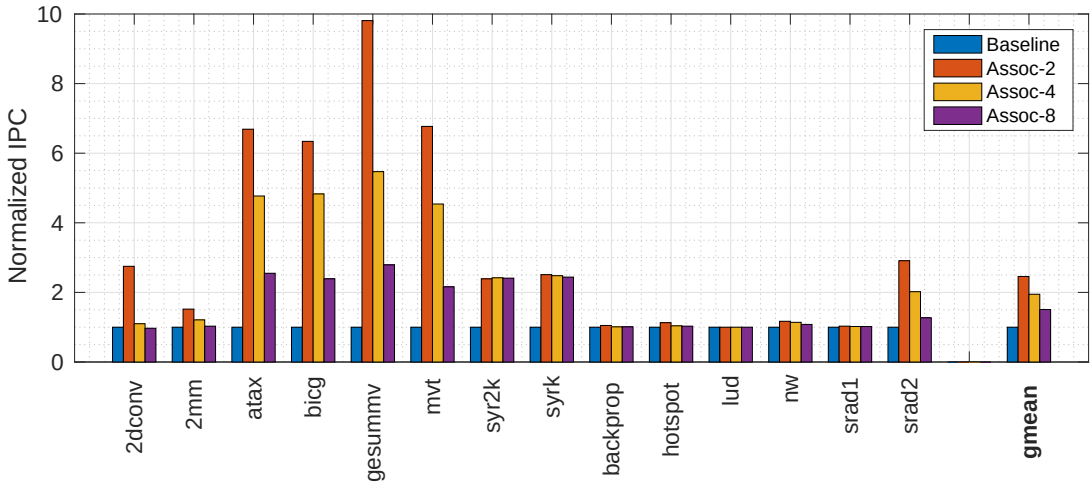


Figure 6.8. IPC improvement with different associativities.

larger associativity cases. Therefore, the IPC improvement for associativity 2 is the most among the three different associativity cases. Cache with associativity 8 still improves the performance by about 1.51x.

6.6 Conclusion

In this chapter, we identified that when the LDST unit is stalled, no other ready warps can probe the cache even if there are potential hits to be found if they could proceed and probe the cache. In order to address this issue, we proposed a memory request scheduling which queues the memory requests from the warp instructions, schedules items in the queue to probe potential hits during LDST unit stall and processes the hit request to efficiently use the cache. The proposed scheme improves the IPC performance by 2.06x over the baseline. It also outperforms the state-of-the-art algorithm, MRPB, by 7%.

CHAPTER 7

RELATED WORK

We introduced chapter-specific related works in earlier chapters. This chapter integrates those with related works to other topics in order to serve as the single central place for all related works to this dissertation.

7.1 Cache Bypassing

7.1.1 CPU Cache Bypassing

Much of the existing research focuses on CPU cache management techniques [28, 38, 39, 46, 71, 85, 87]. Among these, a selection of papers have explored bypassing in CPU caches. Tyson et al. [85] proposed bypassing based on the hit rate of memory access instructions, while Johnson et al. [46] proposed using the access frequency of the cache blocks to predict bypassing. Kharbutli and Solihin [50] proposed using counters of events such as number of references and access intervals to make bypass predictions in the CPU last-level cache. All of these techniques use memory address-related information to make the prediction, costing significant storage overhead that would be impractical for GPU caches.

Program counter trace-based dead block prediction [53] leveraged the fact that sequences of memory instruction PCs tend to lead to the same behavior for different memory blocks. This dead block prediction scheme is useful for making bypass predictions in CPUs. We show that GPU kernels are small, containing only a few distinct memory instructions. Using only the PC to access a block is sufficient for a GPU bypassing prediction.

Cache Bursts [59] is another dead block prediction technique that exploits bursts of accesses hitting the MRU position to improve predictor efficiency. For GPU workloads that use scratch-pad memories, the majority of re-references have been filtered. Gaur et al. [25] proposed bypass and insertion algorithms for exclusive LLCs to adaptively avoid unmodified dead blocks from being written into the exclusive LLC.

7.1.2 GPU Cache Bypassing

Jia et al. [41] justified the effect of the cache in GPUs. They presented a simulation result that L1D cache may degrade the overall system performance. Then, they proposed a static method to analyze GPU programs and determine if caching is beneficial or detrimental at compile time by calculating access stride pattern and applying it to control whether to bypass the L1D cache. Jia et al. [42] later proposed a memory request prioritization buffer (MRPB) to improve GPU performance. MRPB prioritized the memory requests in order to reduce the reuse distance within a warp. It also used cache bypassing to mitigate intra-warp contention. When bypassing, it blindly bypassed memory requests whenever it detects resource contention. Therefore, there are some benchmarks which suffer from performance degradation. Compared to MRPB, our locality-aware selective caching does not degrade performance since we measure the reuse frequency dynamically and conservatively decide caching according to the reuse frequency.

Rogers et al. proposed cache-conscious wavefront scheduling (CCWS) to improve GPU cache efficiency by avoiding data thrashing that causes cache pollution [72]. CCWS estimates working set size of active warps and dynamically restricts the number of warps. This may adversely affect the ability to hide high memory access latency of GPUs. Our locality-aware selective caching bypasses the no-reuse blocks without under-utilizing the SIMD pipeline to reduce cache thrashing.

Lee and Kim proposed a thread-level-parallelism-aware cache management policy to improve performance of the shared last level cache (LLC) in heterogeneous multi-core architec-

ture [55]. They focus on shared LLCs that are dynamically partitioned between CPUs and GPUs. Mekkat et al. proposed a similar idea for heterogeneous LLC management [60], to better partition LLC for GPUs and CPUs in a heterogeneous system.

Li et al. [57] exploit bypassing to reduce contention. They extract locality information during compile time and throttle the warp scheduler to avoid thrashing due to warp contention. Their work focuses on reducing cross-warp contention. Static analysis does not reflect the dynamic behavior of the application.

Tian et al. [80] also exploit the PC to predict bypassing. This method maintains a table for bypass prediction. They use a confidence count to control bypassing. Every reuse decreases the confidence count and every miss increases the confidence count. When the confidence count is greater than the predetermined value for the PC of a memory request, then the request is bypassed. However, this scheme takes a training time until it actually bypasses a request since it uses a confidence counter and the tables are maintained for each L1D cache. To compensate for misprediction, they use a bypassBit in the L2 cache. However, when the bit is set and reset by multiple SM's requests, its bypass decision is not accurate. Our locality-aware selective bypassing maintains a global reuse frequency table to reflect the overall behavior of the program. Also, our scheme dynamically updates the bypass table even by a bypassed request to avoid misprediction.

While the existing works determine bypassing based upon occurred contention or static compile time information, which may lead to be incorrect at runtime, our contention-aware selective caching focuses more on proactively detecting and avoiding cache contention. Also, our work incorporates dynamically obtained locality information to preserve locality.

7.2 Memory Address Randomization

Memory address randomization techniques for CPU caches are well studied. Pseudo-random cache indexing methods have been extensively studied to reduce conflict misses. Topham

et al. [81] use XOR to build a conflict-avoiding cache; Seznec and Bodin [77, 11] combine 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 [54, 69, 70, 91]. Khairy et al. [49] use XOR-based Pseudo Random Interleaving Cache (PRIC) which is used in [81, 70].

Another common approach is to use a secondary indexing method for alternative cache sets when conflicts happen. This category of work includes skewed-associative cache [77], column-associative cache [1], and v-way cache [67].

Some works have also noticed that certain bits in the address are more critical in reducing cache miss rate. Givargis [27] uses off-line 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. [75] propose 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 a direct-mapped cache. ASCIB also needs extra storage to track the changes in the address bits. Wang et al. [86] applied XOR-based index algorithm in their work. It finds that the address range that is more critical to reduce intra-warp cache contention.

The main purpose of the different cache indexing algorithms is to randomly distribute the congested set over all cache sets to minimize the cache thrashing. However, GPGPU executes massively parallelized workloads as much as possible, and requests from different warp can easily overflow the cache sets size. For example, the NVIDIA Fermi architecture uses 16KB L1 data cache which has 128 cache lines (32 sets with 4 lines per set). Just 4 warp's set contending accesses quickly fill up the cache. Indexing may distribute the cache access throughout the whole cache, however, caching the massive amount of cache accesses which may not be used more than once is polluting cache and evicts cache lines which have better locality feature.

7.3 Warp Scheduling

Warp scheduling plays a critical role in sustaining GPU performance and various scheduling algorithms have been proposed based on different heuristics.

Some warp scheduling algorithms use a concurrent throttling technique to reduce contention in an L1D cache. *Static Warp Limiting (SWL)* [72] statically limits the number of warps that can be actively scheduled and needs to be tuned on a per-benchmark basis. *Cache Conscious Warp Scheduling (CCWS)* [72] relies on a dedicated victim cache and a 6-bit Warp ID field in the tag of an 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. *MASCAR* [76] 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 the MSHR has only 1 entry or the queue inside memory part has only 1 slot. On top of CCWS, *Divergence Aware Warp Scheduling (DAWS)* [73] 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. [49] proposed *DWT-CS*, which use core sampling to throttle concurrency. When L1D Miss Per Kilo Instruction (MPKI) is above a given threshold, DWT-CS samples all SMs with a different number of active warps and applies the best-performing active warp count on all SMs.

Some other warp scheduling algorithms are designed to improve GPU resource utilization. Fung et al. [24, 23] investigated the impact of warp scheduling on techniques aiming at branch divergence reduction, i.e., dynamic warp formation and thread block compaction. Jog et al. [45] proposed an orchestrated warp scheduling to increase the timeliness of GPU L1D prefetching. Narasiman et al. [63] 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. [26] introduced another two-level warp scheduler to manage a hierarchical register file design. On top of the two-level warp scheduling, Yu et al. [90] proposed a *Stall-Aware Warp Scheduling (SAWS)* to adjust the fetch group size when pipeline stalls are detected. SAWS mainly focuses on pipeline stalls. Kayiran et al. [48] 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. [56] 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. [44] proposed the *OWL scheduler*, which combines four component scheduling policies to improve L1D locality and the utilization of off-chip memory bandwidth.

The aforementioned warp scheduling techniques do not focus on the problem of LDST stalls and preserving L1D locality especially for cross-warp locality.

7.4 Warp Throttling

Bakhoda et al. [6] present data for several GPU configurations, each with a different maximum number of CTAs that can be concurrently assigned to a core. They observe that some workloads performed better when less CTAs are scheduled concurrently. The data they present is for a GPU without an L1 data cache, running a round-robin warp scheduling algorithm. They conclude that this increase in performance occurs because scheduling less concurrent CTAs on the GPU reduces contention for the interconnection network and DRAM memory system.

Guz et al. [32] use an analytical model to quantify the "performance valley" that exists when the number of threads sharing a cache is increased. They show that increasing the thread

count increases performance until the aggregate working set no longer fits in cache. Increasing threads beyond this point degrades performance until enough threads are present to hide the systems memory latency.

Cheng et al. [18] introduce a thread throttling mechanism to reduce memory latency in multithreaded CPU systems. They propose an analytical model and memory task throttling mechanism to limit thread interference in the memory stage. Their model relies on a stream programming language which decomposes applications into separate tasks for computation and memory and their technique schedules tasks at this granularity.

Ebrahimi et al. [21] examine the effect of disjointed resource allocation between the various components of a chip-multiprocessor system, in particular in the cache hierarchy and memory controller. They observed that uncoordinated fairness-based decisions made by disconnected components could result in a loss of both performance and fairness. Their proposed technique seeks to increase performance and improve fairness in the memory system by throttling the memory accesses generated by CMP cores. This throttling is accomplished by capping the number of MSHR entries that can be used and constraining the rate at which requests in the MSHR are issued to the L2.

These warp throttling techniques do not identify the memory access characteristics of GPU but try to resolve contention by dynamically throttling the number of thread blocks or warps. Our work identifies the memory access characteristic and analyses when caching is beneficial and when not and resolves the contention by reducing the cause of contention.

7.5 Cache Replacement Policy

There is a body of work attempting to increase cache hit rate by improving the replacement or insertion policy [9, 13, 38, 43, 61, 68, 88]. All these attempt to exploit different heuristics of program behavior to predict a blocks re-reference interval and mirror the Belady-optimal [10]

policy as closely as possible.

Li et al. [58] propose Priority Based Cache Replacement (PCAL) policy to tightly couple the thread scheduling mechanism with the cache replacement policy such that GPU cache pollution is minimized while off-chip memory throughput is enhanced. They prioritize the subset of high-priority threads while simultaneously allowing lower priority threads to execute without contending for the cache. By tuning thread-level parallelism while both optimizing cache efficiency as well as other shared resource usage, PCAL improves overall performance. Chen et al. [17] propose G-Cache to alleviate cache thrashing. To detect thrashing, the tag array of L2 cache is enhanced with extra bits (victim bits) to provide L1 cache by some information about the hot lines that have been evicted before. An adaptive cache replacement policy is used by an L1 cache to protect these hot lines. However, the previous works do not incorporate locality information between warps or thread blocks.

CHAPTER 8

CONCLUSION AND FUTURE WORK

8.1 Conclusion

Leveraging the massive computation power of GPUs to accelerate data-intensive applications is a recent trend that embraces the arrival of the big data era. While a throughput processor's cache hierarchy exploits application-inherent locality and can increase the overall performance, the massively parallel execution model of GPUs suffers from cache contention. For applications that are performance-sensitive to caching efficiency, such contention degrades the effectiveness of caches in exploiting locality, thereby suffering from significant performance drop.

This dissertation has categorized the contention into two different categories depending on the source of contention and has examined the memory access request bottlenecks that cause serious cache contention such as memory-divergent instruction caused by column-strided access pattern, cache pollution by no-reuse data blocks, and memory request stall. This dissertation embodies a collection of research efforts to reduce the performance impacts of these bottlenecks from their sources, including contention-aware selective caching, locality-based selective caching, and memory request scheduling. Based on the comprehensive experimental results and systematic comparisons with state-of-the-art techniques, this dissertation has made the following three key contributions:

Contention-aware Selective Caching is proposed to detect the column-strided pattern and its resulting memory-divergent instruction which generates divergent memory accesses, calculates the contending cache sets and locality information, and caches selectively. We demonstrate that

contention-aware selective caching can improve the system more than 2.25x over baseline and reduce memory accesses.

Locality-aware Selective Caching is proposed to detect the locality of memory requests based on per-PC reuse frequency and cache selectively. We demonstrate that the low hardware complexity technique outperforms baseline by 1.39x alone and 2.01x together with contention-aware selective caching, prevents 73% of the no-reuse data from caching and improves reuse frequency in the cache by 27x.

Memory Request Scheduling is proposed to address the memory request stalls at LDST unit. It consists of a memory request schedule queue that holds ready warps' memory requests and a scheduler to effectively schedule them to increase the chances of a hit in the cache lines. We demonstrate that there are 12 ready warps on average when the LDST unit is in a stall and this potential improves the overall performance by 1.95x over baseline and 2.06x along with contention-aware selective caching over baseline.

8.2 Future Work

This dissertation has also opened up opportunities for future architectural research on optimizing the performance of GPU memory subsystem. Particularly, the following two topics are immediate future works.

8.2.1 Locality-Aware Scheduling

From the memory access locality analysis in Section 3.1, the warp scheduler can significantly change the memory access pattern, and thus playing an important role in system performance. Depending on the selection of the scheduler, the overall system performance improvement can be around 20% on average [56]. For some benchmarks, the overall performance is about 3 times better. The technique used in locality-aware selective caching as in Section 5, can be ex-

ploited to schedule the warps to preserve the locality, minimize the eviction from the cache, and also minimize the memory resource contention in the LDST unit. Through the reuse frequency analysis, a dynamic working set can also be calculated to aid the scheduler.

8.2.2 Locality-Aware Cache Replacement Policy

When a cache is full, the new request needs to find an entry to be replaced. The Least Recently Used (LRU) is a commonly used replacement policy. This policy finds a victim by discarding the least recently used item in the cache. This algorithm requires keeping track of what was used and when, which is expensive if one wants to make sure the algorithm always discards the least recently used item. General implementations of this technique require keeping "age bits" for cache-lines and track the "Least Recently Used" cache-line based on the age-bits. Due to the complexity, many implementations of LRU are based on pseudo-LRU.

A GPU thread traffic pattern described in Figure 3.1a may fit with the pseudo-LRU cache replacement policy because of the small working set size, but when warps and thread blocks are involved, pseudo-LRU may no longer efficiently reflect the locality of the GPU memory accesses. However, if the no-reuse blocks are filtered by the locality-based selective caching scheme, the resulting memory access pattern along with LRU policy with reduced insertion and promotion policy, that is, adjusted block insertion and block promotion, can be effectively cached in the L1D. Therefore, cache replacement policy using the locality information developed in Section 5 may improve overall system performance.

PUBLICATION CONTRIBUTIONS

This dissertation has contributed to the following publications. From the thorough analysis of the characteristics of GPU with William Panlener and Dr. Byunghyun Jang, we published a paper titled *Understanding and Optimizing GPU Cache Memory Performance for Compute Workloads* in IEEE 13th International Symposium on Parallel and Distributed Computing (ISPDC) in 2014.

Through cache contention analysis in Chapter 3, we identified the major causes of cache contention. From the first factor of cache contention, column-strided memory traffic patterns and the resulting memory-divergent instruction as introduced in Chapter 4, we submitted a paper, *Contention-Aware Selective Caching to Mitigate Intra-Warp Contention on GPUs* to IISWC 2016. From the second factor, cache pollution caused by caching no reuse data in Chapter 5, we submitted a paper, *Locality-Aware Selective Caching on GPUs* to SBAC-PAD 2016. From the third factor, memory request stall by the non-preemptive LDST unit in Chapter 6, we are preparing a paper titled *Memory Request Scheduling to Promote Potential Cache Hit on GPUs*. I would like to thank my co-authors of the paper, David Troendle, Esraa Abdelmageed, and Dr. Byunghyun Jang for their continuous support, discussion on the idea development, editing and revising.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Agarwal, A., and S. D. Pudar (1993), Column-associative Caches: A Technique for Reducing the Miss Rate of Direct-mapped Caches, *SIGARCH Comput. Archit. News*, 21(2), 179–190, doi:10.1145/173682.165153.

[2] AMD, Inc. (2012), AMD Graphics Cores Next (GCN) Architecture, https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf.

[3] AMD, Inc. (2015), The OpenCL Programming Guide, http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_OpenCL_Programming_User_Guide2.pdf.

[4] Anderson, J. A., C. D. Lorenz, and A. Travesset (2008), General purpose molecular dynamics simulations fully implemented on graphics processing units, *Journal of Computational Physics*, 227(10), 5342 – 5359, doi:http://dx.doi.org/10.1016/j.jcp.2008.01.047.

[5] Bakhoda, A., G. Yuan, W. Fung, H. Wong, and T. Aamodt (2009), Analyzing CUDA workloads using a detailed GPU simulator, in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pp. 163–174, doi:10.1109/ISPASS.2009.4919648.

[6] Bakhoda, A., G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt (2009), Analyzing cuda workloads using a detailed gpu simulator, in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pp. 163–174, doi:10.1109/ISPASS.2009.4919648.

[7] Bakkum, P., and K. Skadron (2010), Accelerating sql database operations on a gpu with cuda, in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, pp. 94–103, ACM, New York, NY, USA, doi:10.1145/1735688.1735706.

[8] Banakar, R., S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel (2002), Scratchpad memory: a design alternative for cache on-chip memory in embedded systems, in *Hardware/-Software Codesign, 2002. CODES 2002. Proceedings of the Tenth International Symposium on*, pp. 73–78, doi:10.1109/CODES.2002.1003604.

[9] Bansal, S., and D. S. Modha (2004), Car: Clock with adaptive replacement, in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST '04, pp. 187–200, USENIX Association, Berkeley, CA, USA.

[10] Belady, L. A. (1966), A study of replacement algorithms for a virtual-storage computer, *IBM Systems Journal*, 5(2), 78–101, doi:10.1147/sj.52.0078.

[11] Bodin, F., and A. Seznec (1997), Skewed associativity improves program performance and enhances predictability, *Computers, IEEE Transactions on*, 46(5), 530–544, doi:10.1109/12.589219.

[12] Brunie, N., S. Collange, and G. Diamos (2012), Simultaneous Branch and Warp Interweaving for Sustained GPU Performance, *SIGARCH Comput. Archit. News*, 40(3), 49–60, doi:10.1145/2366231.2337166.

[13] Chaudhuri, M. (2009), Pseudo-lifo: The foundation of a new family of replacement policies for last-level caches, in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 401–412, doi:10.1145/1669112.1669164.

[14] Che, S., M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron (2009), Rodinia: A benchmark suite for heterogeneous computing, in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pp. 44–54, doi:10.1109/IISWC.2009.5306797.

[15] Chen, L., and G. Agrawal (2012), Optimizing mapreduce for gpus with effective shared memory usage, in *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pp. 199–210, ACM, New York, NY, USA, doi:10.1145/2287076.2287109.

[16] Chen, L., X. Huo, and G. Agrawal (2012), Accelerating mapreduce on a coupled cpu-gpu architecture, in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pp. 25:1–25:11, IEEE Computer Society Press, Los Alamitos, CA, USA.

[17] Chen, X., S. Wu, L.-W. Chang, W.-S. Huang, C. Pearson, Z. Wang, and W.-M. W. Hwu (2014), Adaptive Cache Bypass and Insertion for Many-core Accelerators, in *Proceedings of International Workshop on Manycore Embedded Systems*, MES '14, pp. 1:1–1:8, ACM, New York, NY, USA, doi:10.1145/2613908.2613909.

[18] Cheng, H.-Y., C.-H. Lin, J. Li, and C.-L. Yang (2010), Memory latency reduction via thread throttling, in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pp. 53–64, IEEE Computer Society, Washington, DC, USA, doi:10.1109/MICRO.2010.39.

[19] Choo, K., W. Panlener, and B. Jang (2014), Understanding and Optimizing GPU Cache Memory Performance for Compute Workloads, in *Parallel and Distributed Computing (ISPDC), 2014 IEEE 13th International Symposium on*, pp. 189–196, doi:10.1109/ISPDC.2014.29.

[20] Denning, P. J. (1980), Working sets past and present, *IEEE Trans. Softw. Eng.*, 6(1), 64–84, doi:10.1109/TSE.1980.230464.

[21] Ebrahimi, E., C. J. Lee, O. Mutlu, and Y. N. Patt (2010), Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems, in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pp. 335–346, ACM, New York, NY, USA, doi:10.1145/1736020.1736058.

[22] Fatahalian, K., and M. Houston (2008), A closer look at gpus, *Commun. ACM*, 51(10), 50–57, doi:10.1145/1400181.1400197.

[23] Fung, W., and T. Aamodt (2011), Thread block compaction for efficient SIMT control flow, in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pp. 25–36, doi:10.1109/HPCA.2011.5749714.

[24] Fung, W., I. Sham, G. Yuan, and T. Aamodt (2007), Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow, in *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pp. 407–420, doi:10.1109/MICRO.2007.30.

[25] Gaur, J., M. Chaudhuri, and S. Subramoney (2011), Bypass and insertion algorithms for exclusive last-level caches, in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pp. 81–92.

[26] Gebhart, M., D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron (2011), Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors, *SIGARCH Comput. Archit. News*, 39(3), 235–246, doi:10.1145/2024723.2000093.

[27] Givargis, T. (2003), Improved indexing for cache miss reduction in embedded systems, in *Design Automation Conference, 2003. Proceedings*, pp. 875–880, doi:10.1109/DAC.2003.1219143.

[28] González, A., C. Aliagas, and M. Valero (1995), A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality, in *Proceedings of the 9th International Conference on Supercomputing*, ICS '95, pp. 338–347, ACM, New York, NY, USA, doi:10.1145/224538.224622.

[29] Govindaraju, N., J. Gray, R. Kumar, and D. Manocha (2006), Gputerasort: High performance graphics co-processor sorting for large database management, in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pp. 325–336, ACM, New York, NY, USA, doi:10.1145/1142473.1142511.

[30] Govindaraju, N. K., B. Lloyd, W. Wang, M. Lin, and D. Manocha (2004), Fast computation of database operations using graphics processors, in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pp. 215–226, ACM, New York, NY, USA, doi:10.1145/1007568.1007594.

[31] Grauer-Gray, S., L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos (2012), Auto-tuning a high-level language targeted to GPU codes, in *Innovative Parallel Computing (InPar), 2012*, pp. 1–10, doi:10.1109/InPar.2012.6339595.

[32] Guz, Z., E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. C. Weiser (2009), Many-core vs. many-thread machines: Stay away from the valley, *IEEE Computer Architecture Letters*, 8(1), 25–28, doi:10.1109/L-CA.2009.4.

[33] Hakura, Z. S., and A. Gupta (1997), The Design and Analysis of a Cache Architecture for Texture Mapping, in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pp. 108–120, ACM, New York, NY, USA, doi: 10.1145/264107.264152.

[34] Han, S., K. Jang, K. Park, and S. Moon (2010), Packetshader: A gpu-accelerated software router, *SIGCOMM Comput. Commun. Rev.*, 40(4), 195–206, doi:10.1145/1851275.1851207.

[35] He, B., W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang (2008), Mars: A mapreduce framework on graphics processors, in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pp. 260–269, ACM, New York, NY, USA, doi:10.1145/1454115.1454152.

[36] He, B., M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander (2009), Relational query coprocessing on graphics processors, *ACM Trans. Database Syst.*, 34(4), 21:1–21:39, doi:10.1145/1620585.1620588.

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

[38] Jaleel, A., K. B. Theobald, S. C. Steely, Jr., and J. Emer (2010), High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP), *SIGARCH Comput. Archit. News*, 38(3), 60–71, doi:10.1145/1816038.1815971.

[39] Jalminger, J., and P. Stenstrom (2003), A novel approach to cache block reuse predictions, in *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, pp. 294–302, doi:10.1109/ICPP.2003.1240592.

[40] Jang, B., D. Schaa, P. Mistry, and D. Kaeli (2011), Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures, *IEEE Transactions on Parallel and Distributed Systems*, 22(1), 105–118, doi:10.1109/TPDS.2010.107.

[41] Jia, W., K. A. Shaw, and M. Martonosi (2012), Characterizing and Improving the Use of Demand-fetched Caches in GPUs, in *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pp. 15–24, ACM, New York, NY, USA, doi:10.1145/2304576.2304582.

[42] Jia, W., K. Shaw, and M. Martonosi (2014), MRPB: Memory request prioritization for massively parallel processors, in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pp. 272–283, doi:10.1109/HPCA.2014.6835938.

[43] Jiang, S., and X. Zhang (2002), Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance, *SIGMETRICS Perform. Eval. Rev.*, 30(1), 31–42, doi:10.1145/511399.511340.

[44] Jog, A., O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das (2013), OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance, *SIGPLAN Not.*, 48(4), 395–406, doi:10.1145/2499368.2451158.

[45] Jog, A., O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das (2013), Orchestrated Scheduling and Prefetching for GPGPUs, in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pp. 332–343, ACM, New York, NY, USA, doi:10.1145/2485922.2485951.

[46] Johnson, T. L., D. A. Connors, M. C. Merten, and W. M. W. Hwu (1999), Run-time cache bypassing, *IEEE Transactions on Computers*, 48(12), 1338–1354, doi:10.1109/12.817393.

[47] Katz, G. J., and J. T. Kider, Jr (2008), All-pairs Shortest-paths for Large Graphs on the GPU, in *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '08, pp. 47–55, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland.

[48] Kayiran, O., A. Jog, M. T. Kandemir, and C. R. Das (2013), Neither More nor Less: Optimizing Thread-level Parallelism for GPGPUs, in *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pp. 157–166, IEEE Press, Piscataway, NJ, USA.

[49] Khairy, M., M. Zahran, and A. G. Wassal (2015), Efficient Utilization of GPGPU Cache Hierarchy, in *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*, GPGPU-8, pp. 36–47, ACM, New York, NY, USA, doi:10.1145/2716282.2716291.

[50] Kharbutli, M., and Y. Solihin (2008), Counter-Based Cache Replacement and Bypassing Algorithms, *IEEE Transactions on Computers*, 57(4), 433–447, doi:10.1109/TC.2007.70816.

[51] Khronos OpenCL Working Group (2012), The OpenCL Specification, https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf.

[52] Khronos OpenCL Working Group (2016), The OpenCL Specification, https://www.khronos.org/registry/cl/specs/opencl-2.2.pdf.

[53] Lai, A.-C., C. Fide, and B. Falsafi (2001), Dead-block Prediction &Amp; Dead-block Correlating Prefetchers, in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ISCA '01, pp. 144–154, ACM, New York, NY, USA, doi:10.1145/379240.379259.

[54] Lawrie, D. H., and C. Vora (1982), The Prime Memory System for Array Access, *Computers, IEEE Transactions on*, C-31(5), 435–442, doi:10.1109/TC.1982.1676020.

[55] Lee, J., and H. Kim (2012), TAP: A TLP-aware cache management policy for a CPU-GPU heterogeneous architecture, in *IEEE International Symposium on High-Performance Comp Architecture*, pp. 1–12, doi:10.1109/HPCA.2012.6168947.

[56] Lee, M., S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu (2014), Improving GPGPU resource utilization through alternative thread block scheduling, in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pp. 260–271, doi:10.1109/HPCA.2014.6835937.

[57] Li, A., G.-J. van den Braak, A. Kumar, and H. Corporaal (2015), Adaptive and transparent cache bypassing for GPUs, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 17, ACM.

[58] Li, D., M. Rhu, D. Johnson, M. O'Connor, M. Erez, D. Burger, D. Fussell, and S. Redder (2015), Priority-based cache allocation in throughput processors, in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pp. 89–100, doi:10.1109/HPCA.2015.7056024.

[59] Liu, H., M. Ferdman, J. Huh, and D. Burger (2008), Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency, in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pp. 222–233, IEEE Computer Society, Washington, DC, USA, doi:10.1109/MICRO.2008.4771793.

[60] Mekkat, V., A. Holey, P.-C. Yew, and A. Zhai (2013), Managing Shared Last-level Cache in a Heterogeneous Multicore Processor, in *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pp. 225–234, IEEE Press, Piscataway, NJ, USA.

[61] Meng, J., and K. Skadron (2009), Avoiding cache thrashing due to private data placement in last-level cache for manycore scaling, in *Proceedings of the 2009 IEEE International Conference on Computer Design*, ICCD'09, pp. 282–288, IEEE Press, Piscataway, NJ, USA.

[62] Mosegaard, J., and T. S. Sørensen (2005), Real-time deformation of detailed geometry based on mappings to a less detailed physical simulation on the gpu, in *Proceedings of the 11th Eurographics Conference on Virtual Environments*, EGVE'05, pp. 105–111, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, doi:10.2312/EGVE/IPT_EGVE2005/105-111.

[63] Narasiman, V., M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt (2011), Improving GPU Performance via Large Warps and Two-level Warp Scheduling, in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pp. 308–317, ACM, New York, NY, USA, doi:10.1145/2155620.2155656.

[64] NVIDIA Corporation (2009), NVIDIA Fermi white paper, http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf.

[65] NVIDIA Corporation (2012), NVIDIA Kepler GK110 white paper, https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[66] NVIDIA Corporation (2015), CUDA C Programming Guide, https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[67] Qureshi, M., D. Thompson, and Y. Patt (2005), The V-Way cache: demand-based associativity via global replacement, in *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*, pp. 544–555, doi:10.1109/ISCA.2005.52.

[68] Qureshi, M. K., A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer (2007), Adaptive insertion policies for high performance caching, in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pp. 381–391, ACM, New York, NY, USA, doi:10.1145/1250662.1250709.

[69] Raghavan, R., and J. P. Hayes (1990), On Randomly Interleaved Memories, in *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, Supercomputing '90, pp. 49–58, IEEE Computer Society Press, Los Alamitos, CA, USA.

[70] Rau, B. R. (1991), Pseudo-randomly Interleaved Memory, in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, ISCA '91, pp. 74–83, ACM, New York, NY, USA, doi:10.1145/115952.115961.

[71] Rivers, J. A., E. S. Tam, G. S. Tyson, E. S. Davidson, and M. Farrens (1998), Utilizing Reuse Information in Data Cache Management, in *Proceedings of the 12th International Conference on Supercomputing*, ICS '98, pp. 449–456, ACM, New York, NY, USA, doi: 10.1145/277830.277941.

[72] Rogers, T. G., M. O'Connor, and T. M. Aamodt (2012), Cache-Conscious Wavefront Scheduling, in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pp. 72–83, IEEE Computer Society, Washington, DC, USA, doi:10.1109/MICRO.2012.16.

[73] Rogers, T. G., M. O'Connor, and T. M. Aamodt (2013), Divergence-aware Warp Scheduling, in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pp. 99–110, ACM, New York, NY, USA, doi:10.1145/2540708.2540718.

[74] Roh, L., and W. Najjar (1995), Design of storage hierarchy in multithreaded architectures, in *Microarchitecture, 1995., Proceedings of the 28th Annual International Symposium on*, pp. 271–278, doi:10.1109/MICRO.1995.476836.

[75] Ros, A., P. Xekalakis, M. Cintra, M. E. Acacio, and J. M. García (2012), 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 '12, pp. 51–56, ACM, New York, NY, USA, doi:10.1145/2333660.2333674.

[76] Sethia, A., D. Jamshidi, and S. Mahlke (2015), Mascar: Speeding up GPU warps by reducing memory pitstops, in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pp. 174–185, doi:10.1109/HPCA.2015.7056031.

[77] Seznec, A. (1993), A Case for Two-way Skewed-associative Caches, in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pp. 169–178, ACM, New York, NY, USA, doi:10.1145/165123.165152.

[78] Stuart, J. A., and J. D. Owens (2011), Multi-gpu mapreduce on gpu clusters, in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pp. 1068–1079, IEEE Computer Society, Washington, DC, USA, doi:10.1109/IPDPS.2011.102.

[79] Ta, T., K. Choo, E. Tan, B. Jang, and E. Choi (2015), Accelerating DynEarthSol3D on tightly coupled CPUGPU heterogeneous processors, *Computers & Geosciences*, 79, 27 – 37, doi: http://dx.doi.org/10.1016/j.cageo.2015.03.003.

[80] Tian, Y., S. Puthoor, J. L. Greathouse, B. M. Beckmann, and D. A. Jiménez (2015), Adaptive GPU cache bypassing, in *Proceedings of the 8th Workshop on General Purpose Processing using GPUs*, pp. 25–35, ACM.

[81] Topham, N., A. González, and J. González (1997), The Design and Performance of a Conflict-avoiding Cache, in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 30, pp. 71–80, IEEE Computer Society, Washington, DC, USA.

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

[83] Trancoso, P., D. Othonos, and A. Artemiou (2009), Data parallel acceleration of decision support queries using cell/be and gpus, in *Proceedings of the 6th ACM Conference on Computing Frontiers*, CF '09, pp. 117–126, ACM, New York, NY, USA, doi:10.1145/1531743.1531763.

[84] Trapnell, C., and M. C. Schatz (2009), Optimizing Data Intensive GPGPU Computations for DNA Sequence Alignment, *Parallel Comput.*, 35(8-9), 429–440, doi:10.1016/j.parco.2009.05.002.

[85] Tyson, G., M. Farrens, J. Matthews, and A. R. Pleszkun (1995), A Modified Approach to Data Cache Management, in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, MICRO 28, pp. 93–103, IEEE Computer Society Press, Los Alamitos, CA, USA.

[86] Wang, B., Z. Liu, X. Wang, and W. Yu (2015), Eliminating Intra-warp Conflict Misses in GPU, in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pp. 689–694, EDA Consortium, San Jose, CA, USA.

[87] Wierzbicki, A., N. Leibowitz, M. Ripeanu, and R. Wozniak (2004), Cache Replacement Policies Revisited: The Case of P2P Traffic.

[88] Wu, C.-J., A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, Jr., and J. Emer (2011), Ship: Signature-based hit predictor for high performance caching, in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pp. 430–441, ACM, New York, NY, USA, doi:10.1145/2155620.2155671.

[89] Wu, H., G. Diamos, S. Cadambi, and S. Yalamanchili (2012), Kernel weaver: Automatically fusing database primitives for efficient gpu computation, in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 107–118, doi:10.1109/MICRO.2012.19.

[90] Yu, Y., W. Xiao, X. He, H. Guo, Y. Wang, and X. Chen (2015), A Stall-Aware Warp Scheduling for Dynamically Optimizing Thread-level Parallelism in GPGPUs, in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pp. 15–24, ACM, New York, NY, USA, doi:10.1145/2751205.2751234.

[91] Zhang, Z., Z. Zhu, and X. Zhang (2000), A Permutation-based Page Interleaving Scheme to Reduce Row-buffer Conflicts and Exploit Data Locality, in *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 33, pp. 32–41, ACM, New York, NY, USA, doi:10.1145/360128.360134.

VITA

Education

2016:	Ph.D. in Computer Science	*University of Mississippi*, University, MS
2001:	M.S. in Electrical Engineering Systems	*University of Michigan*, Ann Arbor, MI
2000:	B.S. in EE and CS	*Handong Global University*, Pohang, Korea

Professional Experience

May-Aug 2015:	Software engineering intern. *Google Inc.*, Mountain View, CA
May-Aug 2014:	Research intern. *Samsung Research America*, San Jose, CA
2014-2015:	Graduate instructor. *University of Mississippi*, University, MS
2012-2016:	Graduate research assistant. *University of Mississippi*, University, MS
2002-2012:	Senior engineer / manager. *Samsung Electronics*, Suwon, Korea
2000-2001:	Graduate research assistant. *University of Michigan*, Ann Arbor, MI

Publication List

Papers

1. K. Choo, W. Panlener, and B. Jang,
   Understanding and optimizing GPU cache memory performance for compute workloads, Parallel and Distributed Computing (ISPDC), 2014 IEEE 13th International Symposium on, pages 189-196, IEEE, 2014.

2. T. Ta, K. Choo, E. Tan, B. Jang, and E. Choi,
   Accelerating DynEarthSol3D on tightly coupled CPUGPU heterogeneous processors, Computers & Geosciences, vol. 79, pp. 27-37, Jun. 2015.

3. K. Choo, D. Troendle, E. Abdelmageed, and B. Jang,
   Contention-Aware Selective Caching to Mitigate Intra-Warp Contention on GPUs, submitted to IISWC 2016 (from Chapter 4).

4. K. Choo, D. Troendle, E. Abdelmageed, and B. Jang,
   Locality-Aware Selective Caching on GPUs, submitted to SBAC-PAD 2016 (from Chapter 5).

5. K. Choo, D. Troendle, E. Abdelmageed, and B. Jang,
   Memory request scheduling to promote potential cache hit on GPU, to be submitted (from Chapter 6).

6. D. Troendle, K. Choo, and B. Jang,
Recency Rank Tracking (RRT): A Scalable, Configurable, Low Latency Cache Replace-ment Policy, to be submitted.

Patents

1. E. Park, J.H. Lee, and K. Choo
Digital transmission system for transmitting additional data and method thereof. US8,891,674, 2014.

2. S. Park, H.J. Jeong, S.J. Park, J.H. Lee, K. Kim, Y.S. Kwon, J.H. Jeong, G. Ryu, K. Choo, and K.R. Ji
Digital broadcasting transmitter, digital broadcasting receiver, and methods for configuring and processing a stream for same. US 8,891,465, 2014.

3. G. Ryu, Y.S. Kwon, J.H. Lee, C.S. Park, J. Kim, K. Choo, K.R. Ji, S. Park, and J.H. Kim
Digital broadcast transmitter, digital broadcast receiver, and methods for configuring and processing streams thereof. US 8,811,304, 2014.

4. J.H. Jeong, H.J. Lee, S.H. Myung, Y.S. Kwon, K.R. Ji, J.H. Lee, C.S. Park, G. Ryu, J. Kim, and K. Choo
Digital broadcasting transmitter, digital broadcasting receiver, and method for composing and processing streams thereof. US 8,804,805, 2014.

5. K. Choo and J.H. Lee
OFDM transmitting and receiving systems and methods. US 8,804,477, 2014.

6. Y.S. Kwon, G. Ryu, J.H. Lee, C.S. Park, J. Kim, K. Choo, K.R. Ji, S. Park, J.H. Kim
Digital broadcast transmitter, digital broadcast receiver, and methods for configuring and processing streams thereof. US 8,798,138, 2014.

7. Y.S. Kwon, G. Ryu, J.H. Lee, C.S. Park, J. Kim, K. Choo, K.R. Ji, S. Park, and J.H. Kim
Digital broadcast transmitter, digital broadcast receiver, and methods for configuring and processing digital transport streams thereof. US 8,787,220, 2014.

8. G. Ryu, S. Park, J.H. Kim, and K. Choo
Method and apparatus for transmitting broadcast, method and apparatus for receiving broadcast. US 8,717,961, 2014.

9. J.H. Lee, K. Choo, K. Ha, H.J. Jeong
Service relay device, service receiver for receiving the relayed service signal, and methods thereof. US 8,140,008, 2012.

10. E. Park, J. Kim, S.H. Yoon, K. Choo, K. Seok
Trellis encoder and trellis encoding device having the same. US 8,001,451, 2011.

Honors and Awards

- Graduate Student Achievement Award, Spring 2016, University of Mississippi
- President of $\Upsilon\Pi E$ (UPE), CS Honor Society, Fall 2015 - Spring 2016, University of Mississippi Chapter
- Doctoral Dissertation Fellowship Award, Spring 2016, University of Mississippi
- Computer Science SAP Scholarship Award, Spring 2016, University of Mississippi
- UPE Scholarship Award, Fall 2015, $\Upsilon\Pi E$ (UPE)
- Academic Excellence Award (2nd place in class of 2000), 2000, Handong Global University
- 4-year full Scholarship, 1996 - 2000, Handong Global University