ABSTRACT

DAI, HONGWEN. GPU Memory Architecture Optimization. (Under the direction of Dr. Huiyang Zhou).

General purpose computation on graphics processing units (GPGPU) has become prevalent in high performance computing. Besides the massive multithreading, GPUs have adopted multi-level cache hierarchies to mitigate the long off-chip memory access latency. However, significant cache trashing and severe memory pipeline stalls exist due to the massive multithreading. In this dissertation, we study the inefficiencies of memory request handling in GPUs and propose architectural designs to improve performance for both single kernel execution and concurrent kernel execution.

First, I will present our work of a model-driven approach to GPU cache bypassing. In this work, we propose a simple yet effective performance model to estimate the number of cache hits and reservation failures due to cache-miss-related resource congestion as a function of the number of warps/thread-blocks to access/bypass the cache. Based on the model, we design a cost-effective hardware-based scheme to identify the optimal number of warps/ thread-blocks to bypass the L1 D-cache. The key difference from prior works on GPU cache bypassing is that we do not rely on accurate prediction on hot cache lines. Compared to warp throttling, we do not limit the number of active warps so as to exploit the available thread-level parallelism (TLP) and otherwise underutilized NoC (Network on Chip) and off-chip memory bandwidth. Furthermore, our scheme is simple to implement and it does not alter the existing cache organization.

Second, I will illustrate our study on a sound baseline in GPU memory architecture research. We thoroughly investigate the performance impact of different design choices and suggest a set of sound configurations to use in future studies. First, we show that advanced cache indexing functions greatly reduce conflict misses and improve cache performance on GPUs. Then, we demonstrate that among the two cache line allocation policies, allocate-on-fill brings a better performance than allocate-on-miss. Third, we show that the number of MSHRs (Miss Status Holding Registers) is an important design factor to explore. Fourth, we demonstrate that Modulo mapping of memory partitions may cause severe partition camping,

resulting in underutilization of DRAM bandwidth and the capacity of banked L2 cache. On the other hand, Xor mapping can greatly mitigate the problem of memory partition camping. Furthermore, we show the fact that a realistic number of in-flight bypassed requests can be supported should be taken into account in GPU cache bypass study to achieve more reliable results and conclusions.

Third, I will present our schemes to accelerate concurrent kernel execution (CKE) in GPUs. We show that although it is potential for intra-SM sharing to better utilize resources within an SM, the negative interference among kernels may undermine the overall performance. Specifically, as concurrent kernels share the memory subsystem, one kernel, even as computing-intensive, may starve from not being able to issue memory requests in time. Besides, severe L1 D-cache thrashing and memory pipeline stalls caused by one kernel, especially a memory-intensive one, will impact other kernels, further hurting the overall performance. In this study, we investigate various schemes to overcome the aforementioned problems exposed in intra-SM sharing. We first highlight that cache partitioning techniques proposed for CPUs are not effective for GPUs. Then we propose two schemes to reduce memory pipeline stalls. The first one is to balance memory accesses from individual kernels. The second one is to limit the number of inflight memory instructions issued from individual kernels. Our proposed schemes significantly improve system throughput and fairness of two state-of-the-art intra-SM sharing schemes, Warped-Slicer and SMK with lightweight hardware overheads.

GPU Memory Architecture Optimization

by
Hongwen Dai

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina

2017

APPROVED BY:

Dr. Huiyang Zhou Committee Chair	Dr. Eric Roternberg
Dr. James Tuck	Dr. Xipeng Shen

DEDICATION

To my wife Jingru Yang, my parents,

family, teachers and friends.

BIOGRAPHY

Hongwen Dai was born in Yibin, a beautiful city in Sichuan province in China. He has obtained his bachelor degree in 2009 and master degree in 2012 in Electrical Engineering from Zhejiang University, Hangzhou, China.

He started his PhD program in Computer Engineering in 2012 at North Carolina State University (NCSU). Under the supervision of Dr. Huiyang Zhou, he has worked on several research projects related to computer architecture research focusing on GPU memory architecture optimizations. He has done internships with ORNL and Apple while he was a student. He is a member of IEEE, ACM and IEEE Computer Society.

ACKNOWLEDGMENTS

This dissertation would not have been possible without the help and support of a number of people. First and foremost, I would like to thank my PhD advisor, Dr. Huiyang Zhou for the tremendous time, effort and wisdom that he has invested in my graduate study. His insightfulness on research, evidence oriented style and constant curiosity towards science and technology will be treasured by me for a long time.

I would like to thank Dr, Eric Roternberg, Dr. James Tuck and Dr. Xipeng Shen for serving on my PhD advisory committee and providing their critiques as well as encouragement on my research projects. I also would like to show my appreciation to Dr. Chao Li, Dr. Saurabh Gupta, Dr. Christos Kartsaklis, Chen Zhao and Michael Mantor for their great suggestions and collaborations on my research.

I had brilliant colleagues at NCSU and many thanks go to Chao Li, Saurabh Gupta, Zhen Lin, Qi Jia, Xiangyang Guo, Yuan Liu, Ping Xiang and Yi Yang for their help during my study. I especially want to thank Chao Li, Saurabh Gupta and Zhen Lin for their valuable advices on my research.

Also, I would like to express my thanks to my friends Yang Li, Youyang Zheng, Hang Xu, Chenghao Deng, Lizhong Chen and Chao Li (SJTU) for their friendship and encouragement during my journey of graduate study.

I would like to thank my wife Jingru Yang, my father Dakui Dai and my mother Xiaoping Bai deeply for their continuous support, trust and love.

Finally, I would like to dedicate this dissertation to all who have helped, supported and loved me in my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

General purpose computation on graphics processing units (GPGPU) has become prevalent in high performance computing. Modern GPUs consist of multiple Streaming Multiprocessors (SMs), each of which contains many computing units and multiple schedulers. A GPU kernel is launched with a grid of thread blocks (TBs). Threads within a TB form multiple warps and all threads in a warp execute in a lock step manner. Every cycle, each warp scheduler makes the decision on which of its supervised warps to execute next, according to the applied warp scheduling policy.

Besides the massive multithreading, GPUs have adopted multi-level cache hierarchies to mitigate the long off-chip memory access latency. However, the cache capacity per thread and the cache line lifetime on GPUs are much smaller than those on CPUs, resulting in significant cache trashing. Moreover, since miss status handling registers (MSHRs) and miss queue entries need to be allocated for outstanding misses, massive multithreading can also cause severe memory pipeline stalls on GPUs when such resources are fully occupied. Simply enlarging cache capacity and/or adding more cache-miss-related resources is costly and sometimes impractical due to the required area and power.

Considering the performance impact of cache thrashing and cache-miss-related resource congestion, we propose a simple yet effective performance model to estimate the number of cache hits and reservation failures due to cache-miss-related resource congestion as a function of the number of warps/thread-blocks to access/bypass the cache. Based on the model, we design a cost-effective hardware-based dynamic scheme to identify the optimal number of warps/thread-blocks to bypass the L1 D-cache. The key difference from prior works on GPU cache bypassing is that we do not rely on accurate prediction on hot cache lines. Compared to warp throttling, we do not limit the number of active warps so as to exploit the available thread-level parallelism (TLP) and otherwise underutilized NoC (Network on Chip)

and off-chip memory bandwidth. Furthermore, our scheme is simple to implement and it does not alter the existing cache organization.

Besides, although people have been using the cycle-level micro-architecture simulator GPGPUSim [5] for their studies on GPU memory architecture, a sound baseline has not been constructed, and this may lead to pathological performance results and inaccurate conclusions. Thus we thoroughly investigate the performance impact of different design choices and suggest a set of sound baseline configurations to use in future studies. First, we show that advanced cache indexing functions can greatly reduce conflict misses and improve cache performance on GPUs. Then, we demonstrate that among the two cache line allocation policies to deal with cache-miss-related resources allocation for outstanding misses, allocate-on-fill achieves a better performance than allocate-on-miss due to more cache hits and fewer reservation failures since allocate-on-fill preserves data longer in cache before eviction and requires fewer resources for an outstanding miss. Third, since the allocated MSHRs are reserved until the required data comes back from lower memory levels, it is intuitive to boost performance by deploying more MSHRs to reduce reservation failures and thus memory pipeline stalls. However, our experiments show that with the interleaved warp execution, more MSHRs may lead to more warps scheduled to access the L1 D-cache and increase the possibility of cache thrashing, resulting in degraded performance. Fourth, Modulo mapping among memory partitions may cause severe partition camping, resulting in underutilization of DRAM bandwidth and the capacity of banked L2 cache. And our experiments show that Xor mapping can greatly mitigate the problem of memory partition camping. Furthermore, prior works [7][23][34][36][72][10] on GPU cache bypass have not discussed the necessary hardware structure to store the relevant information of bypassed requests but just assumed there are always adequate hardware resources to store such information so that unlimited number of in-flight bypassed requests can be supported. However, such an assumption is too optimistic and in practical, just a limited number of bypassed requests can be supported, depending on the size of the supporting hardware structure. So we also investigate how prior GPU bypassing schemes perform with such constraints and conclude the fact that a realistic number of in-flight

bypassed requests can be supported should be taken into account in GPU cache bypass study, to achieve more reliable results and conclusions.

Moreover, following the advances in technology scaling, GPUs incorporate an increasing amount of computing resources. As a consequence, it becomes difficult for a single GPU kernel to fully utilize the vast GPU resources. One solution to improve resource utilization is concurrent kernel execution (CKE), which enables multiple GPU kernels to share a GPU concurrently. Early CKE mainly targets at the leftover resources, which are not utilized by a single kernel. However, it fails to optimize the resource utilization and does not provide fairness among concurrent kernels. Spatial multitasking assigns a subset of streaming multiprocessors (SMs) to each kernel. Although achieving better fairness, the resource underutilization within an SM is not addressed. Therefore, intra-SM sharing has been proposed to issue thread blocks from multiple kernels to each SM. However, as shown in our study, the overall performance may be undermined even in the state-of-the-art intra-SM sharing schemes, due to the interference among concurrent kernels. Specifically, as concurrent kernels share the memory subsystem, one kernel, even as computing-intensive, may starve from not being able to issue memory instructions in time. Besides, severe L1 D-cache thrashing and memory pipeline stalls caused by one kernel, especially a memory-intensive one, will impact other kernels, further hurting the overall performance. We investigate various schemes to overcome the aforementioned problems exposed in intra-SM sharing. We first highlight that cache partitioning techniques proposed for CPUs are not effective for GPUs. Then we propose two schemes to reduce memory pipeline stalls. The first scheme is to balance memory accesses from individual kernels. The second one is to limit the number of inflight memory instructions issued from individual kernels. Our evaluation shows that the proposed schemes significantly improve the system throughput and fairness of two state-of-the-art intra-SM sharing schemes, Warped-Slicer and SMK with lightweight hardware overheads.

Chapter 2

A Model-Driven Approach to Warp/Thread-Block Level GPU Cache Bypassing

2.1 Introduction

General purpose computation on graphics processing units (GPGPU) has become prevalent in high performance computing. Modern GPUs have adopted multi-level cache hierarchies to mitigate the long off-chip memory access latency. However, the cache capacity per thread and the cache line lifetime on GPUs are much smaller than those on CPUs, resulting in significant cache trashing especially due to inter-warp contention [52]. Moreover, since miss status handling registers (MSHRs) and miss queue entries need to be allocated for outstanding misses, massive multithreading on GPUs can also cause severe memory pipeline stalls when such resources are fully occupied. Simply enlarging cache capacity and/or adding more cache-miss-related resources is costly and sometimes impractical due to the required area and power.

Thread throttling [7][17][36][52][72] has been proposed based on the observation that intra-warp locality is crucial for highly cache sensitive workloads. Cache Conscious Wavefront Scheduling (CCWS) [52] improves performance by limiting the number of actively scheduled warps, thereby reducing L1 D-cache thrashing and preserving intra-warp locality.

However, latency hiding via massive multithreading is affected by warp throttling due to the reduced number of active warps. Furthermore, warp throttling may cause Network-on-Chip (NoC) and DRAM bandwidth to be underutilized. Cache bypassing [34][13][59] has been applied to protect ‘hot’ cache lines from early eviction. However, with so many threads sharing caches on GPUs, it is difficult to make robust predictions on hot cache lines. Coordinated Bypassing and Warp Throttling (CBWT) [7] combines protection distance based cache

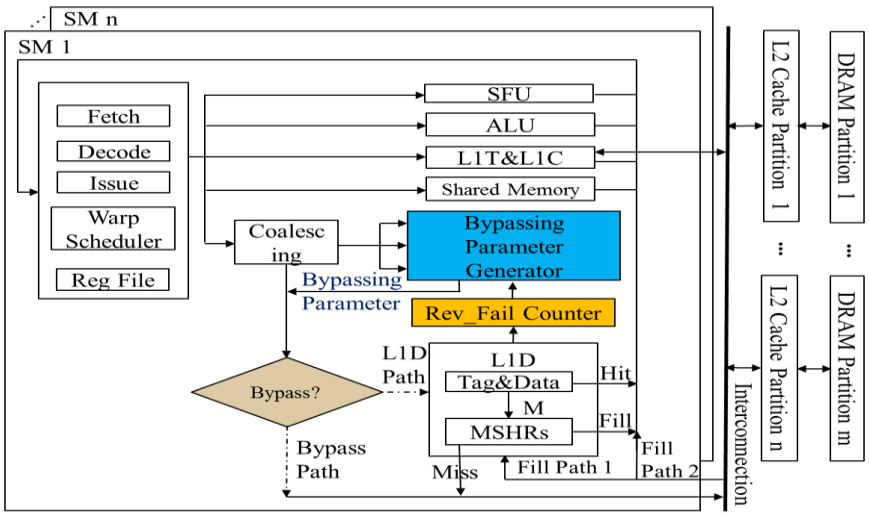


Figure 1. Baseline GPU. Our proposed dynamic scheme adds a Bypassing Parameter Generator and minor change in the bypassing decision logic.

bypassing (PDP) [11] and warp throttling to overcome the limitations of cache-bypassing-only or warp-throttling-only solutions.

In this paper, we propose a simple yet effective performance model to estimate the number of cache hits and reservation failures due to cache-miss-related resource congestion as a function of the number of warps/thread-blocks (TBs) to access/bypass the cache. Based on the model, we design a cost-effective hardware-based dynamic scheme to find the optimal number of warps/TBs to bypass the L1 D-cache. The key difference from prior works on GPU cache bypassing is that we do not rely on accurate prediction on hot cache lines. Compared to warp throttling, we do not limit the number of active warps so as to exploit the available thread-level parallelism (TLP) and otherwise underutilized NoC and off-chip memory bandwidth. Furthermore, our scheme is simple to implement and it does not alter the existing cache organization. Overall, this paper makes the following contributions:

(1) We characterize cache behaviors of GPGPU applications and show that cache capacity contention and cache-miss-related resource congestion should be considered jointly for GPU cache bypassing.

(2) We propose a simple yet effective performance model to estimate the number of cache hits and reservation failures to find the optimal number of warps/ thread-blocks to bypass the L1 D-cache. Based on the model, we design a dynamic GPU cache bypassing scheme.

(3) Our scheme achieves significant performance improvements over the baseline and outperforms the state-of-the-art GPU cache management techniques, including CCWS and CBWT.

2.2 Background and Related Work

2.2.1 Baseline Architecture

A GPU kernel is launched with a grid of thread blocks (TBs). Threads within a TB form multiple warps and all threads in a warp execute in a lock step manner. One GPU consists of multiple Streaming Multiprocessors (SMs). As shown in ***Figure 1***, on each SM, there are a L1 read-only texture cache and a constant cache, a data cache (L1 D-cache) and shared memory. A unified L2 cache is shared among multiple SMs. Typically, the L1 D-cache uses write-through with either write-allocate [1] or write-no-allocate [43][42] policies, and the L2 cache uses the write-back write-allocate to save the DRAM bandwidth [57].

2.2.2 Baseline Memory Request Handling

On GPUs, global and local memory requests from threads in a warp are coalesced into as few transactions as possible before being sent to the memory hierarchy. The cached or bypassed information is typically encoded in instruction opcodes [45], indicating whether a request is sent to the L1 D-cache through the 'L1D path' or directly sent to the L2 cache through the 'Bypass Path', as shown in ***Figure 1***. For a request sent to the L1 D-cache, if it is a miss, the resources including a cache line slot, a MSHR entry and a miss queue entry need to be allocated. If any of these cache-miss-related resources is not available, a reservation failure occurs and the memory pipeline is stalled. The MSHR entry is reserved until the data is fetched from the L2 cache/off-chip memory while the miss queue entry is released when the miss request is forwarded to the L2 cache. Compared to bypassed requests, a request send to the L1 D-cache enjoys low access latency if it hits in the L1 D-cache. However, massive requests sent

to the L1 D-cache can easily cause cache thrashing and cache-miss-related resource congestion, thereby degrading the overall performance.

In our warp/TB level cache bypassing scheme, once a warp/TB is determined to bypass the L1 D-cache, all its accesses go to the 'Bypass Path'. This is straightforward to implement, compared to the schemes which predict and bypass zero/low reused cache lines.

2.2.3 Related Work

Guz et al. [17] demonstrate that increasing the number of threads accessing a cache can improve performance until the aggregate working set no longer fits in the cache. Kayiran et al. [28] and Xie et al. [72] dynamically adjust the number of TBs accessing L1 D-caches. Rogers et al. [52] propose CCWS to control the number of actively scheduled warps. In comparison, our scheme adaptively works on either the warp or TB level and identify the optimal number of warps/TBs to access the L1 D-cache. The rest warps/TBs are not suspended, and instead they simply bypass the L1 D-cache to leverage the otherwise underutilized NoC and memory bandwidth. In other words, we do not penalize TLP like thread throttling approaches.

On GPU cache management, Jia et al. propose MRPB[23] to preserve intra-warp locality and bypasses the cache when there are memory pipeline stalls. Detecting and protecting hot cache lines has also been proposed [34][59]. However, irregular memory access patterns make accurate detection challenging.

Recent works have also exploited the combination of thread throttling and cache bypassing. Li et al. [36] propose priority-based cache allocation on top of CCWS. Chen et al. propose CBWT [7] that adopts PDP for L1 D-cache bypassing and applies warp throttling if the contention on the L2 cache and DRAM is severe. However, PDP is not as effective as it is on CPUs. Li et al [33] propose a compile-time framework for cache bypassing at the warp level for global memory reads.

The aforementioned works either require multiple profiling runs [7][33][36][52] or incur non-trivial hardware overheads due to reorder queues[23], warp throttling detection [7] [52][72], finite state machines [36], and cache line protection schemes [7][34][59]. In contrast,

Table 1. Baseline architecture configuration

# of SMs	15, SIMD width=32, 1.4GHz
Per-SM warp schedulers	2 Greedy-Then-Oldest warp schedulers
Per-SM limit	1536 threads, 48 warps, 8 thread blocks, 32 MSHRs, 8 miss queue entries
Per-SM L1D-cache	16KB, 128B line, 4-way associativity
Per-SM shared memory	48KB, 32 banks
Unified L2 cache	768 KB, 128KB/partition, 6 partitions, 128B line, 16-way associativity
L1D/L2 policies	alloc-on-miss, LRU, L1D:WTWN, L2: WBWA
Interconnect	32B channel width, 1.4GHz
DRAM	6 memory channels, FR-FCFS scheduler, 924MHz, BW: 48bytes/cycle

our sampling-based scheme is much simpler to implement and the results in Section 2.7 show that it can achieve higher performance than the state-of-the-art techniques.

Several GPU performance models have been proposed. Hong et al. [20] use memory warp parallelism and computation warp parallelism to estimate the performance. Zhang et al. [74] develop a micro-benchmark based performance model to measure the execution time of the instruction pipeline, shared memory accesses, and global memory accesses. Huang et al. [21] propose GPUMech, which profiles the instruction trace of every warp and models multithreading and resource contentions caused by memory divergence. In comparison, our model captures cache contention and cache-miss-related resource congestion as a function of the number of warps/TBs accessing the cache, which has not been addressed in prior models.

2.3 Experimental Methodology

Simulation Environment: We use GPGPUSim V3.2.1 [5], a cycle-accurate GPU microarchitecture simulator, to evaluate our proposed scheme. The baseline GPU architecture configuration is shown in ***Table 1***, based on the Fermi architecture. To reduce the conflict misses, we experimented different cache indexing functions [29][14] and choose to use the bitwise-XOR indexing function as baseline for its simplicity and effectiveness.

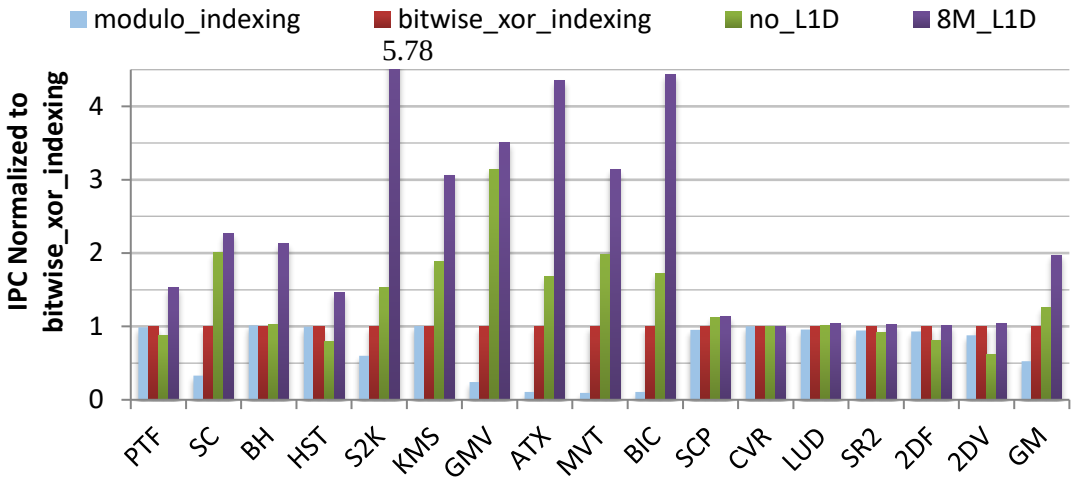


Figure 2. Performance under different L1 D-cache configurations.

Benchmarks: We evaluate a collection of benchmarks, listed in ***Table 2***, from Rodinia [8], Polybench [15] and LonestarGPU [6], including both regular and irregular applications. HST and SCP are the data cache version adopted from [35]. We evaluate all these workloads with their default grid/block dimensions/inputs, scaled if necessary, similar to prior works [23][39]. All simulations run to completion.

2.4 Characterization and Motivation

We first analyse the performance impact of the L1 D-cache. ***Figure 2*** shows the performance under different L1 D-cache configurations, including: (1) a 16KB L1 D-cache with the default modulo cache indexing, (2) a 16KB L1 D-cache with the bitwise-XOR cache indexing, (3) no L1 D-cache, and (4) an 8MB L1 D-cache.

We make the following observations from ***Figure 2***. First, it demonstrates that the indexing function has a remarkable performance impact. On average, the bitwise-XOR indexing function has almost 2X speedup over the modulo indexing function. Second, compared to that when there is no L1 D-cache, the 16KB L1 D-cache improves the performance of the benchmarks PTF, HST, SR2, 2DF and 2DV. On the other hand, many other benchmarks, including SC, S2K, KMS etc., show better performance when there is no L1 D-cache. The reason is that they do not suffer from cache-miss-related resource congestion and

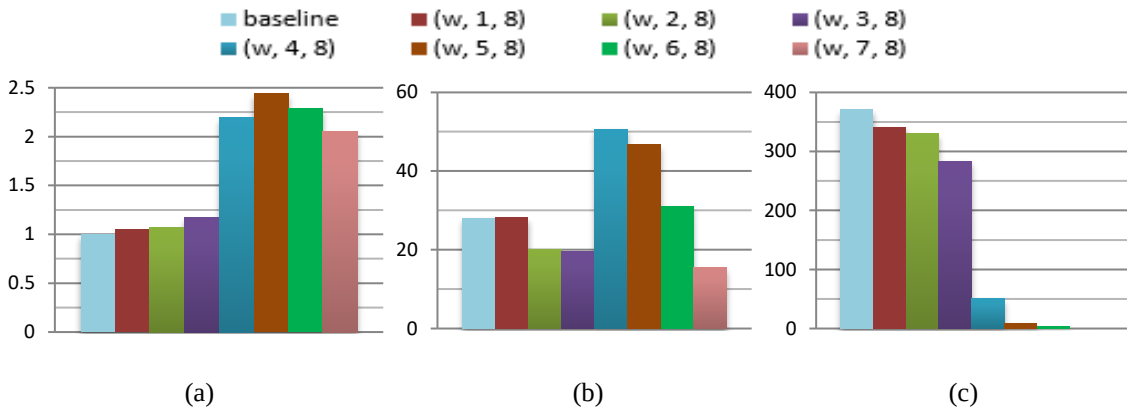
Table 2. Benchmarks

Name	Description	Type	Suite
PTF	ParticleFilter	HCC	[8]
SC	StreamCluster	HCC	[8]
BH	Barnes-Hut	HCC	[6]
HST	Histogram	HCC	[35]
S2K	Symmetric rank-2k operations	HCC	[15]
KMS	K-means clustering	HCC	[8]
GMV	Scalar-vector-matrix multiply	HCC	[15]
ATX	Matrix-transpose-vector multiply	HCC	[15]
MVT	Matrix-vector-product transpose	HCC	[15]
BIC	BiCGStab linear solver subkernel	HCC	[15]
SCP	ScalarProduct	LCC	[35]
CVR	Covariance Computation	LCC	[15]
LUD	LU Decomposition	LCC	[8]
SR2	Srad2	LCC	[8]
2DF	2D Finite different time domain	LCC	[15]
2DV	2D Convolution	LCC	[15]

the resulting memory pipeline stalls. Third, an 8MB L1 D-cache greatly improves the performance by reducing both cache capacity contention and cache-miss-related resource congestion.

Based on the performance improvement obtained from an 8MB L1 D-cache, we classify the benchmarks into two categories: High Cache Contention (HCC) ones with more than 50% performance improvement over the baseline and Low Cache Contention (LCC) ones with less than 50% improvement (***Figure 2*** and ***Table 2***). Although we focus on HCC benchmarks, the results of LCC ones are included to show the robustness of our scheme.

Next, we look into warp/TB level GPU cache bypassing as the intra-warp locality is preserved. We use **(w, M, N)** to denote bypassing *M* of *N* warps; and **(b, M, N)** for bypassing *M* of *N* TBs. An interesting case study on the benchmark KMS is shown in ***Figure 3***. ***Figure 3***(a) shows the performance initially increases and then decreases with more warps being bypassed. The reason is that although the L1 D-cache contention is further relieved with more



(a) (b) (c)

Figure 3. Bypassing various numbers of warps for KMS: (a) Normalized IPC, (b) L1 D-cache hits per Kilo-Inst and (c) Reservation Failures per Kilo-Inst (RFKI).

warps bypassed, the cache utilization is affected. As shown in ***Figure 3***(b), (w, 4, 8) has more hits than (w, 5, 8). On the other hand, (w, 4, 8) incurs more reservation failure than (w, 5, 8) as shown in ***Figure 3***(c). The combined effect leads to the fact that (w, 5, 8) achieves the best performance. From this case study, we conclude that a metric integrating both cache hits and reservation failures is needed to determine the optimal number of warps/TBs to access/bypass the cache.

2.5 A Performance Model Integrating Cache Contention and Related Resource Congestion

In this section, we present our model to determine how many warps/TBs to bypass the L1 D-cache. First, we use the number of cache hits instead of hit rates to model cache contention. Then, we propose an empirical model to estimate the impact of cache-miss-related resource congestion. Next, we define a new '*Adjusted Hits*' metric integrating both cache hits and reservation failures. The optimal number of warps/TBs to bypass the L1 D-cache is the one maximizing this new metric.

2.5.1 Modelling Cache Contention

Although the cache hit rate shows how often the cached data is reused, it does not correlate well with the overall performance because it is oblivious to the number of accesses, which may

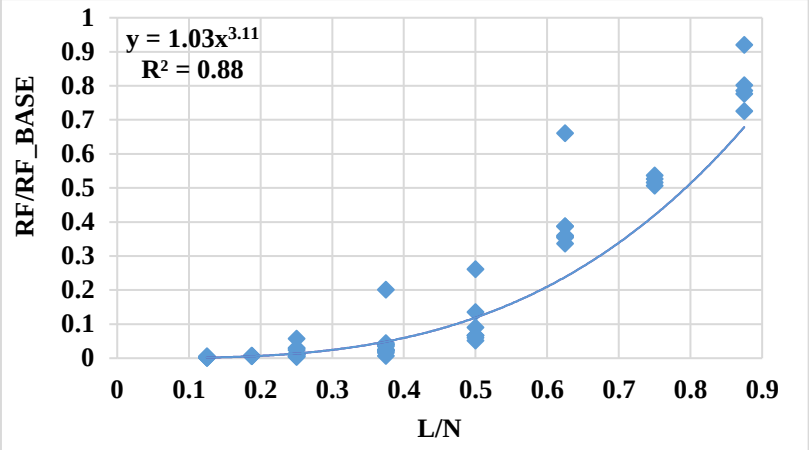


Figure 4. Identifying the relationship between the (normalized) reservation fails and the (normalized) number of warps/TBs accessing the L1 D-cache.

vary due to bypassing. Therefore, we use the number of hits to measure the performance impact of cache contention.

2.5.2 Modelling Cache-Miss-Related Resource Congestion

To model cache-miss-related resource congestion, we propose an empirical model capturing the relationship between the number of reservation failures and the number of bypassed warps/TBs.

Our empirical model is constructed using curve fitting, similar to the studies on cache miss rates vs. cache capacity [9][18]. To this end, we run simulations with various numbers of warps/TBs bypassing the L1 D-cache for the 5 HCC benchmarks from Polybench [15], and collect the numbers of reservation failures. Specifically, if there are more than one TB running on an SM, we vary the number of TBs to bypass the L1 D-cache; if there is only one TB on an SM, we vary the number of warps. Assume the total number of reservation failures on an SM is *RF_base* for the baseline, i.e. when all N warps/TBs access the L1 D-cache. The goal of our model is to calculate *RF_estimated*, the estimated number of reservation failures when *M* warps/TBs bypass the L1 D-cache, i.e., when *L* warps/TBs access the L1 D-cache, where *L*= *N-M*. ***Figure 4*** shows that the number of reservation failures (normalized to *RF_base*) varies with the number of warps/TBs (normalized to *N*) accessing the L1 D-cache. Then, we perform

curve fitting and identify a cubic relationship between (*RF_estimated*/*RF_base*) and (*L*/*N*). The coefficient of determination, R^2, is 0.88, indicating the formula is indicative enough to use as our goal is to match the trend rather than reproducing the exact number. Based on the cubic relationship, we estimate the number of reservation failures as following:

$$RF\_estimated = RF\_base * (L/N)^3$$

2.5.3 Putting It All Together

The goal of optimal bypassing is to maximize cache hits while minimizing cache-miss-related resource congestion. To achieve this, we propose a new metric '*Adjusted Hits*' to incorporate both the number of hits and reservation failures:

$$Adjusted\ Hits = Hits\_estimated - RF\_estimated*E$$

Here *Hits_estimated* is obtained from sampling in our dynamic bypassing scheme. As described in Section 2.5.2, we profiled 5 HCC benchmarks from Polybench and found that when E is in the range of 0.3 to 0.7, the metric '*Adjusted Hits*' correlates well with the overall performance (i.e. IPC). Therefore, we choose E as 0.5 as it can be implemented with a shift operation. The optimal bypassing parameter is the one maximizing '*Adjusted Hits*'.

2.6 Dynamic Warp/TB Level Bypassing

Based on the empirical model presented in Section 2.5, we propose a bypassing parameter generator (BPG). ***Figure 5*** shows the organization of a BPG, which has a set of '*Adjusted Hits*' calculators and a bypassing parameter selector.

An '*Adjusted Hits*' calculator estimates the number of hits and reservation failures. It uses an online monitor to compute the number of hits. The online monitor is essentially a shadow tag array, which has the same structure as the tag store in the L1 D-cache. We adopt set sampling [49] to mitigate the hardware cost. In our implementation, 4 out of 32 sets (i.e. sampling ratio 1:8) are sampled. An '*Adjusted Hits*' calculator obtains the number of reservation failures from the single global reservation failure counter as shown in ***Figure 1*** and then uses the model described in Section 2.5.2 to calculate *RF_estimated* when bypassing a

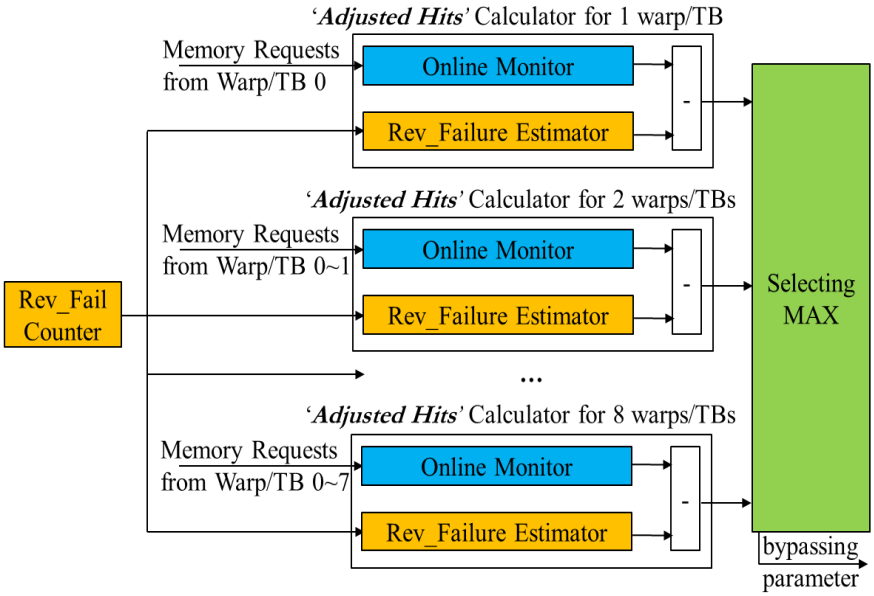


Figure 5. The organization of a BPG.

specific number of warps/TBs. This module is implemented with the Rev_Failure Estimator shown in Figure ***Figure 5***.

Each '*Adjusted Hits*' calculator is dedicated to tracking a specific number of warps/TBs. For instance, the first '*Adjusted Hits*' calculator generates the '*Adjusted Hits*' for warp/TB 0, the second one works for warps/TBs 0~1 and so on. In a BPG, there is a set of 8 '*Adjusted Hits*' calculators supporting 1 to 8 warps/TBs. The reason is that there can be up to 8 TBs on one SM on the Fermi architecture and generally no more than 8 warps (256 threads) within a TB. Although 8 '*Adjusted Hits*' calculators seem to be limiting when there are more than 8 warps in a TB (e.g. PTF has 16 warps in one TB), our experiments show the optimal number of warps/TBs accessing L1 D-cache is smaller than 8 when there is opportunity for bypassing. So we keep the hardware overhead small by limiting the number of calculators to 8 in a BPG.

Considering the complexity of the cubic operation and the small range of bypassing parameters (8 in total), each Rev_Failure Estimator is implemented as a 1-entry pre-computed lookup table for $(L/N)^3$ and a multiplier for $RF\_base * (L/N)^3$.

After the '*Adjusted Hits*' calculators produce '*Adjusted Hits*' for different numbers of warps/TBs accessing the L1 D-cache, we select the one, L_{max}, that results in the highest '*Adjusted Hits*', and have the first L_{max} warps/TBs access L1 D-cache while bypassing the rest. If the BPG figures out that it is best to have 8 warps/TBs access L1 D-cache, it means no opportunity for warp/TB bypassing and all warps/TBs continue to access L1 D-cache.

In our experiment, a new bypassing parameter is produced every 1000 accesses from all warps/TBs on one SM. The sampling interval of 1000 accesses works well in capturing the phase behaviour. In order to not completely lose the history information, the hit counters and the reservation failure counter are right shifted by one bit every time a new bypassing parameter is created.

We use CACTI 6.5 [66] to evaluate the hardware cost of BPG. The majority source of area overhead is the shadow tag arrays used in '*Adjusted Hits*' calculators. Each shadow tag array entry is 35-bits (1 valid bit + 2 LRU bits + up to 32 bits tag). With 8 '*Adjusted Hits*' calculators on each SM, the total area for shadow tag arrays is estimated as 0.087mm^2 for 15 SMs using the 45nm technology. Such additional area is approximately 0.016% of the GTX480 area [42], which is a 15-SM system with the 40nm technology. Other hardware costs include one 10-bit hit number counter for each '*Adjusted Hits*' calculator, one global 12-bit reservation failure counter on each SM, etc. Those miscellaneous overheads are negligible in comparison. Since there is one BPG per SM, we refer to this design as local BPG or L-BPG.

In the L-BPG design, the hardware overhead is proportional to the number of SMs. To reduce the hardware cost, we propose a global BPG design, G-BPG, which monitors application behaviors on one SM and applies the bypassing parameter to other SMs. The bypassing parameter broadcasting is not on the critical path and can be done with simple logic added to the existing TB-dispatcher. As shown in ***Figure 6***, G-BPG is able to reap most performance improvement of L-BPG.

We also use CACTI 6.5 [66] to evaluate the energy efficiency of the shadow tag arrays. The dynamic read energy per access and total leakage power is estimated as 0.001 nJ and 0.27

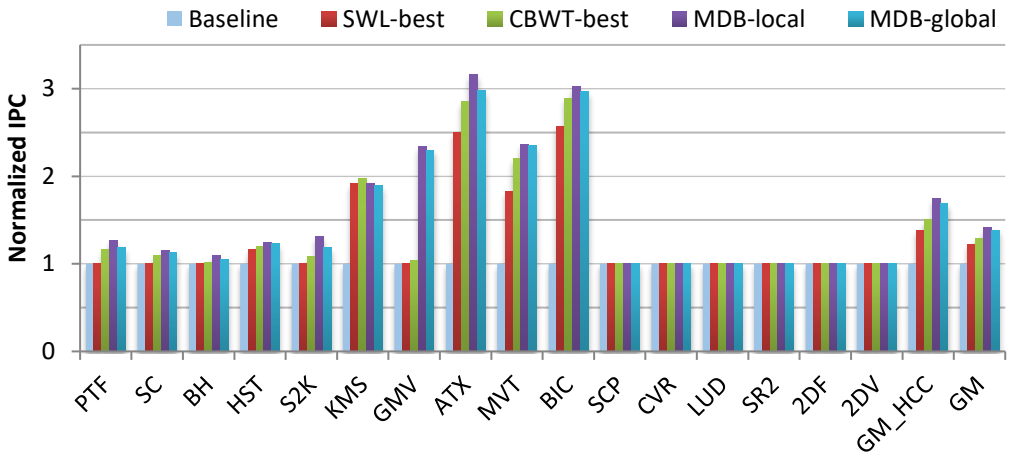


Figure 6. Performance of different GPU cache management schemes.

mW, respectively. Such small energy overhead is negligible compared to the substantial static energy savings from the significantly reduced execution time.

2.7 Experimental Results and Analysis

In this section, we conduct experimental analysis on our bypassing schemes and compare them with state-of-the-art techniques.

2.7.1 Performance Evaluation and Analysis

In the evaluation, we compare our Model-Driven Bypassing (MDB) scheme to two closely related techniques. All performance results are normalized to the baseline, which sends all memory requests to L1 D-cache (i.e. no bypassing). The bitwise-XOR mapping function is used for both L1 D-caches and the L2 cache.

SWL-best: The optimal Static Wavefront Limiting (SWL) configuration. As demonstrated in the prior work [52], SWL-best outperforms dynamic CCWS due to the start-up cost associated with the latter.

CBWT-best: The optimal SWL with PDP bypassing enabled. As demonstrated by Chen et al.[7], CBWT outperforms pure bypassing/warp-throttling and CBWT-best performs better than dynamic CBWT.

MDB-local: Our proposed dynamic warp/TB level GPU bypassing scheme, which deploys one BPG per SM.

MDB-global: Our proposed dynamic warp/TB level GPU bypassing scheme, which adopts the global BPG design.

a) Performance Comparison.

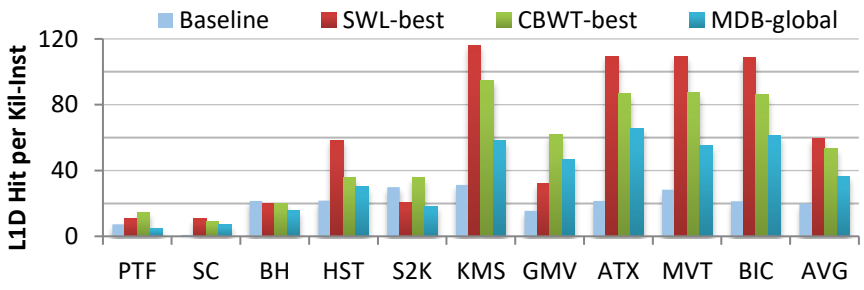
Figure 6 shows performance of different GPU cache management schemes. First, it shows the effectiveness of MDB. Although the model (Section 2.5.2) is derived from curve fitting among a subset of HCC benchmarks, both MDB-local and MDB-global improve the performance of all HCC benchmarks and show no performance degradation for the LCC benchmarks. We also confirm that the cubic relationship, identified in Section 2.5.2, holds in general for all HCC benchmarks. GM_HCC represents the average (geometric mean) performance of HCC benchmarks in ***Figure 6***.

Comparing MDB-global with MDB-local, we observe that MDB-global can reap most of the benefits of the latter with significantly less hardware overhead. MDB-global achieves an average of 1.69x speedup over the baseline, close to the 1.75x speedup from MDB-local. It shows a mild performance loss compared to MDB-local due to the diverse runtime behaviors on different SMs, for the benchmarks such as PTF, S2K, etc. In the following discussion, we focus on MDB-global.

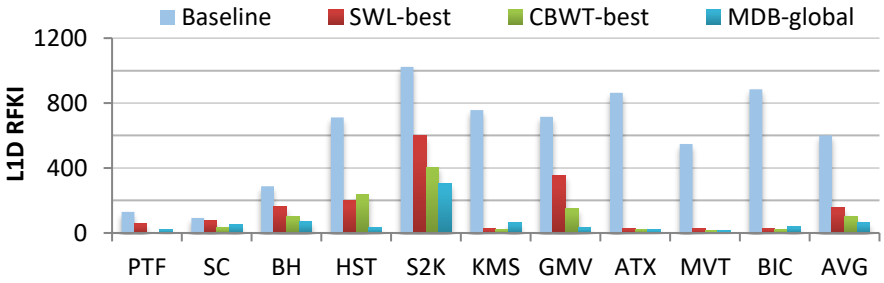
Figure 6 also demonstrates that our MDB approach outperforms the prior GPU cache management schemes. While SWL-best and CBWT-best achieve an average of 1.39x and 1.51x speedup over the baseline, respectively, MDB-global achieves an average of performance improvement of 21.6% over SWL-best and 11.9% over CBWT-best. Next, we dissect the reasons. We focus on HCC benchmarks as all these schemes have small impact on LCC ones.

b) Hits & Reservation Failures per Kilo-Instruction

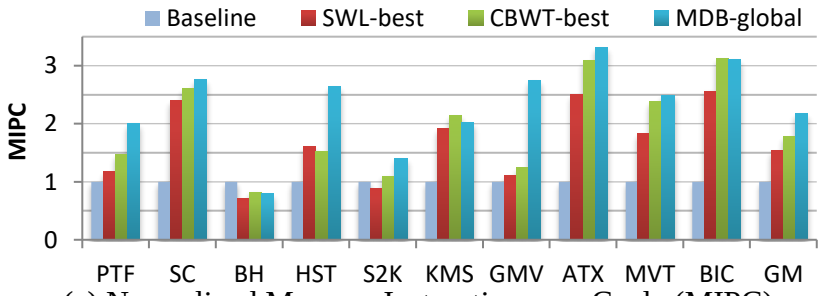
As discussed in Section 2.2.2, both cache hits and reservation failures due to cache-miss-related resource congestion affect the overall performance. Here, we report the number of



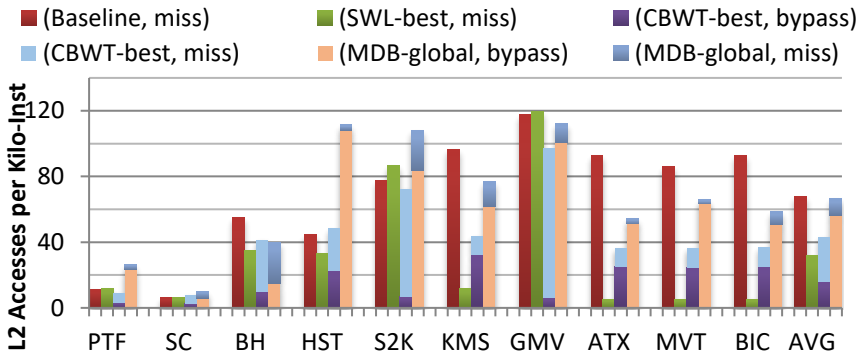
(a) L1 D-cache hits per Kilo-Inst



(b) L1 D-cache Reservation Failures per Kilo-Inst (RFKI)



(c) Normalized Memory Instructions per Cycle (MIPC)



(d) L2 accesses per Kilo-Inst, sourced from L1 D-cache bypassed accesses and misses

Figure 7. L1 D-cache performance and L1-L2 traffic.

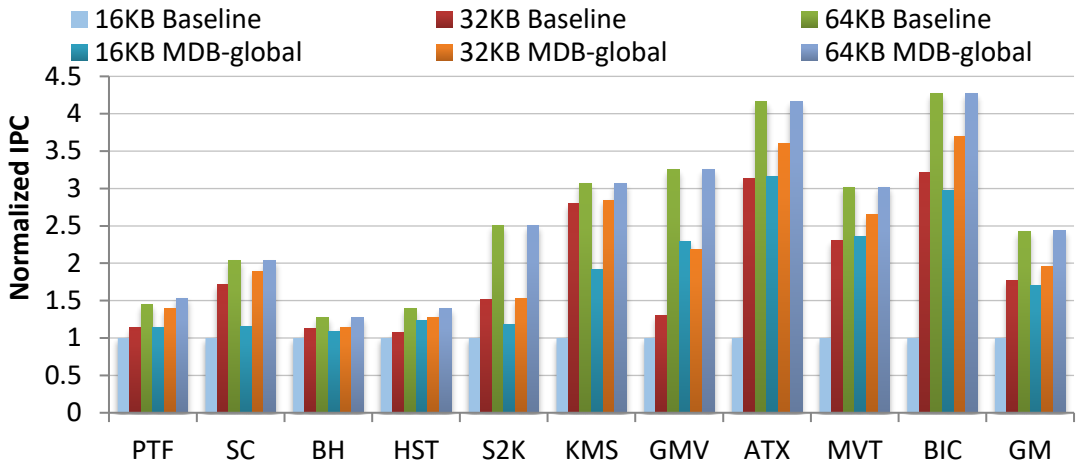
cache hits and reservation failures in ***Figure 7*** (a) and (b). We make several observations from

the figure. First, SWL-best significantly increases the number of hits and reduces the number of reservation failures for HST, KMS, ATX, MVT and BIC while having minor impact on other benchmarks, leading to higher performance. Second, CBWT-best is more effective than SWL-best by combining warp throttling and the PDP cache bypassing scheme. CBWT-best improves the performance of all HCC benchmarks. For those benchmarks where SWL-best has minor impact, CBWT-best either brings more hits, e.g., for PTF, S2K and GMV, or has fewer reservation failures, e.g. for SC. MDB-global approach explicitly models reservation failures, thereby being more effective than CBWT-best. On average, MDB-global has the lowest reservation failures. MDB-global experiences more reservation failures than CBWT-best on KMS because there is a start-up delay in MDB-global and KMS is sensitive to the timing of bypassing.

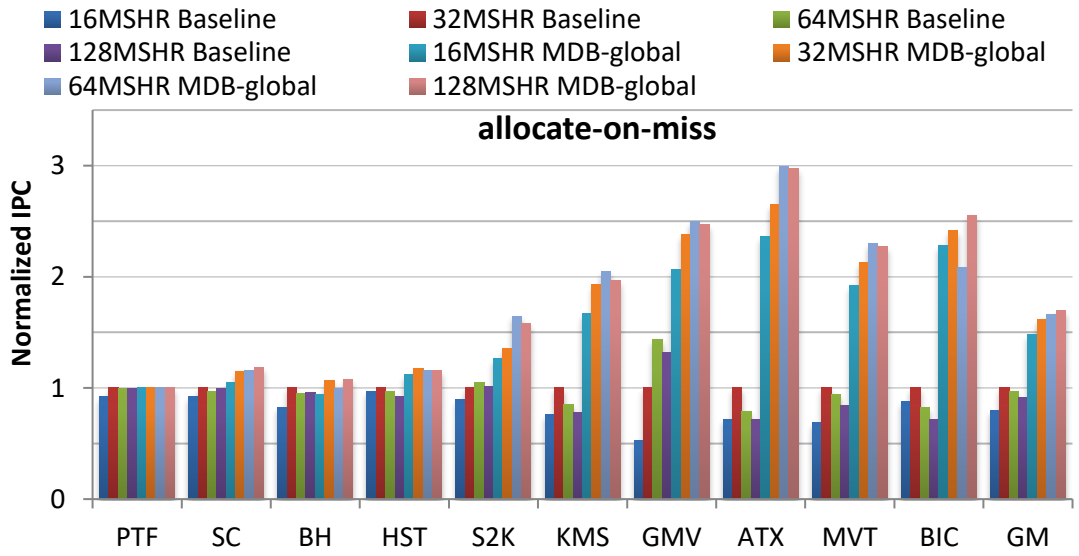
c) MIPC, L1-L2 Traffic, and L2-DRAM Traffic

Since both the "L1D Path" and "Bypass Path" can be used to serve data, we use MIPC (Memory Instructions served Per Cycle) to capture the complete picture of memory subsystem performance, as shown in ***Figure 7***(c). First, we can see that the trend of MIPC matches the overall performance. Second, although SWL-best has high hits and low reservation failures per kilo-instruction, it performs worse than CBWT-best and MDB because it sacrifices TLP and MLP (memory-level parallelism). In addition to warp throttling, CBWT-best adopts PDP to relieve cache contention and leverages the underutilized NoC and DRAM bandwidth. In this way, CBWT-best has a higher MIPC than SWL-best. Compared to SWL-best and CBWT-best, our MDB scheme does not necessarily have higher hits per kilo-instructions, but it shows lower RFKI and higher MIPC on average due to aggressive bypassing.

Figure 7(d) shows the L1-L2 traffic, sourced from L1 D-cache misses and bypassed accesses. SWL-best reduces the L1-L2 traffic from the improved L1 D-cache efficiency for KMS, ATX, etc. Compared to SWL-best, CWBT-best shows heavier L1-L2 traffic on KMS, ATX and so on, lighter on S2K and GMV, and similar on PTF, depending on the combined effect of higher L1 D-cache efficiency and cache bypassing. MDB-global reduces L2 traffic



(a) Performance with different L1 D-cache capacities, normalized to a 16KB L1 D-cache.



(b) Performance with different MSHR sizes, normalized to a 16KB L1 D-cache with 32MSHR.

Figure 8. Sensitivity study.

for KMS, GMV and so on but shows heavier L2 traffic for PTF, HST and S2K. The majority of L2 traffic of MDB-global is from bypassed accesses. The fact that MDB-global is aggressive on cache bypassing leads to its effectiveness in preserving intra-warp locality, improving cache efficiency, reducing cache-miss-related resource congestion, and achieving higher

performance. In addition, SWL-best, CBWT-best and our MDB approach all achieve lighter DRAM traffic, compared to the baseline.

2.7.2 Sensitivity Study

1) Sensitivity to L1 D-cache Capacity

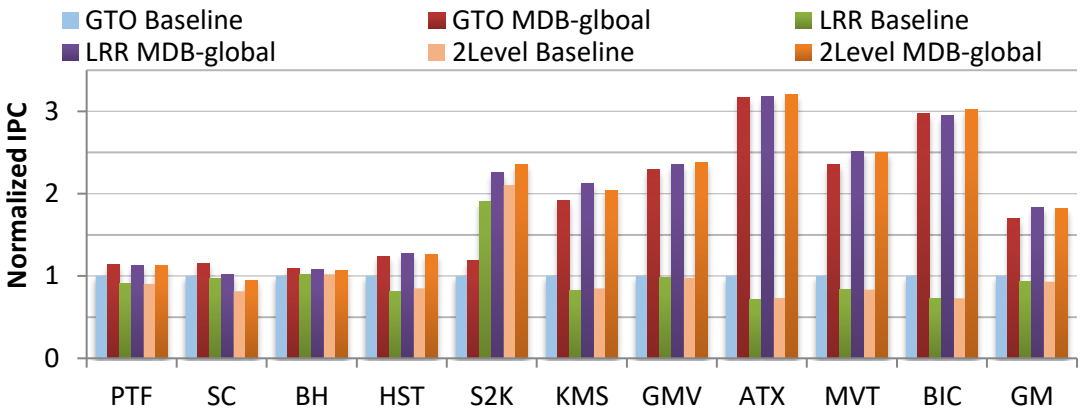
Figure 8(a) shows the sensitivity of MDB-global on different L1 D-cache capacities. MDB-global shows performance improvement for each cache capacity, even the large ones. On the other hand, with a larger cache capacity, the performance improvement is reduced. On average, the normalized performance from a base 16KB, 32KB and 64KB L1 D-cache is 1x, 1,77x and 2.43x, and MDB-global boosts the performance to 1.70x, 1.96x and 2.44x, respectively. Moreover, we can see a 16KB L1 D-cache with our proposed MDB-global can achieve similar performance to a 32KB L1 D-cache. This highlights the cost/area effectiveness of our approach, compared to enlarging the L1 D-cache capacity.

2) Sensitivity to L1 D-cache MSHR Size

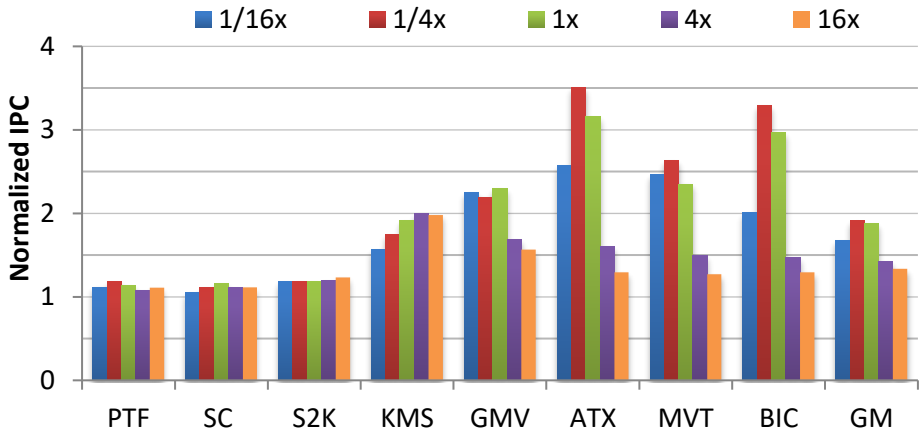
Figure 8(b) shows the sensitivity of MDB-global to MSHR sizes on a 16KB L1 D-cache. The examined MSHR size is from the small 16MSHR to the large 128MSHR. As shown, the performance shows an up-then-down trend with more MSHRs under for some benchmarks (KMS, ATX, etc.) because the combined effect of MSHR throttling and warp scheduling policy re-shapes memory access trace and may lead to the degradation of L1 D-cache efficiency. Nevertheless, MDB-global consistently improves the performance with different MSHRs. On Average, our proposed MDB-global can double the performance with any of the examined MSHR sizes. This implies directly adding more MSHRs is not only costly in terms of power and area but also not effective to improve performance and an intelligent warp scheduling or cache management scheme is necessary.

3) Sensitivity to Warp Scheduling Policy

All the experiments discussed so far use the Greedy-Then-Oldest (GTO) warp scheduling policy, which runs a single warp until a long latency operation and then picks up the oldest one. GTO improves performance by preserving intra-warp locality and performs better than



(a) Performance under different warp scheduling policies, normalized to GTO.



(b) Performance improvement with different input sizes.

Figure 9. Sensitivity study (Cont.).

Two-Level warp scheduling [39] and Round-Robin (RR) policies. Nevertheless, ***Figure 9***(a) shows that MDB-global continuously improves the performance despite the variations when different warp scheduling policies are applied. On average, the normalized performance from a base GTO, LRR and 2Level is 1x, 0.93x and 0.94x, and MDB-global boosts the performance to 1.70x, 1.83x and 1.82x, respectively.

4) Sensitivity to application input size

Figure 9(b) presents the sensitivity of MDB-global to application input size. Six benchmarks, of which the input size can be easily configured with a specific ratio, are shown in the figure. The examined input sizes include 1/16x, 1/4x, 1x, 4x and 16x of the size used in prior

experiments. As shown, MDB-global constantly improves the performance with various input sizes of applications. On average, MDB-global achieves 1.68x, 1.91x, 1.87x, 1.42x and 1.34x performance over that from the baseline speedup for the examined benchmarks with 1/16x, 1/4x, 1x, 4x and 16x of the previously used input size, respectively.

2.8 Conclusion

Throughput-oriented GPGPUs hide long operation latency with massive multithreading. However, limited per thread cache capacity and massive memory requests can easily cause cache thrashing and memory pipeline stalls. In this paper, we propose a performance model for L1 D-cache contention and cache-miss-related resource congestion. Based on the model, we design a cost-effective dynamic warp/TB level GPU cache bypassing scheme. The experimental results show that our scheme achieves significant performance improvement over the baseline and outperforms the state-of-the-art GPU cache management schemes. We also demonstrate that our scheme remains effective with different cache capacities, varied number of MSHRs, various warp scheduling policies and altered application input sizes.

Chapter 3

The Demand for a Sound Baseline GPU Cache Design in GPU Memory Architecture Research

3.1 Introduction

General purpose computation on graphics processing units (GPGPU) has become prevalent in high performance computing. Modern GPUs consist of multiple Streaming Multiprocessors (SMs), each of which containing 32 to 192 CUDA cores and 2 to 4 warp schedulers [43][44][42][46]. A GPU kernel is launched with a grid of thread blocks (TBs). Threads within a TB form multiple warps and all threads in a warp execute in a lock step manner.

Besides massive multithreading, GPUs have adopted multi-level cache hierarchies to mitigate long off-chip memory access latencies and there have been significant research works on GPU memory architecture. In this work, we highlight several often-overlooked aspects of GPU cache design as well as the request distribution among memory partitions. We show the remarkable performance impact from these aspects (up to 7x), which may well overshadow the performance impact from a newly proposed memory architecture design. Thus, we argue that a sound baseline should be deployed in GPU memory architecture research.

Cache-set indexing. Although cache indexing methods have been well studied to reduce conflict misses in CPUs [29][14][38], no prior works have thoroughly studied the performance impact of various advanced cache indexing functions on GPUs.

Cache-line allocation. The cache allocation policy, allocate-on-miss vs. allocate-on-fill, has significant impact on GPUs due to the relatively short lifetime of cache lines. For a request sent to the L1 D-cache, if a miss, cache-miss-related resources are allocated before the request is forwarded to the L2 cache. With allocate-on-miss, a cache line slot, a MSHR, and a miss queue entry need to be allocated. In contrast, with allocate-on-fill, a MSHR and a miss queue entry need to be allocated when an outstanding miss occurs but the victim cache line

slot is chosen when the required data has returned from lower memory levels. In either policy, if any of the required resources is not available, a reservation failure occurs and the memory pipeline is stalled. The allocated MSHR is reserved until the data is fetched from the L2 cache/off-chip memory while the miss queue entry is released once the miss request is forwarded to the lower memory hierarchy. Since allocate-on-fill preserves the victim cache line longer in the cache before eviction and reserves fewer resources for an outstanding miss, it tends to enjoy more cache hits and fewer reservation failures, and in turn better performance than allocate-on-miss. Although allocate-on-fill requires extra buffering and flow-control logics to fill data to the cache in-order, the in-order execution model and the write-evict policy make the GPU L1 D-cache friendly to allocate-on-fill as there is no dirty data to write to L2 when a victim cache line is to be evicted at the fill time. Therefore, it is intriguing to investigate how well allocate-on-fill performs for GPGPU applications.

MSHR sizes. Since the allocated MSHRs are reserved until the required data are returned, it is intuitive to boost performance by deploying more MSHRs to reduce reservation failures and thus memory pipeline stalls, and to increase memory-level-parallelism (MLP). However, more MSHRs may lead to more warps scheduled to access the L1 D-cache and the interconnect network in a short interval, increasing the possibility of cache thrashing or network congestion. So it is useful to better understand the performance impact of the MSHR size.

Memory-partition mapping. In GPUs, memory requests are distributed among multiple memory partitions. Although the Modulo address mapping is simple to implement and effective for some applications, it may result in severe memory partition camping, in which requests are disproportionately handled by a small subset of memory partitions, leading to the underutilization of DRAM bandwidth and the capacity of the banked L2 cache. Therefore, it is important to examine how memory request distribution among partitions and the overall performance will be affected by different memory partition mapping functions.

Limited bypassing requests. Prior works [7][10][23][34][36][72] on GPU cache bypassing assume that there are always adequate hardware resources to store the relevant

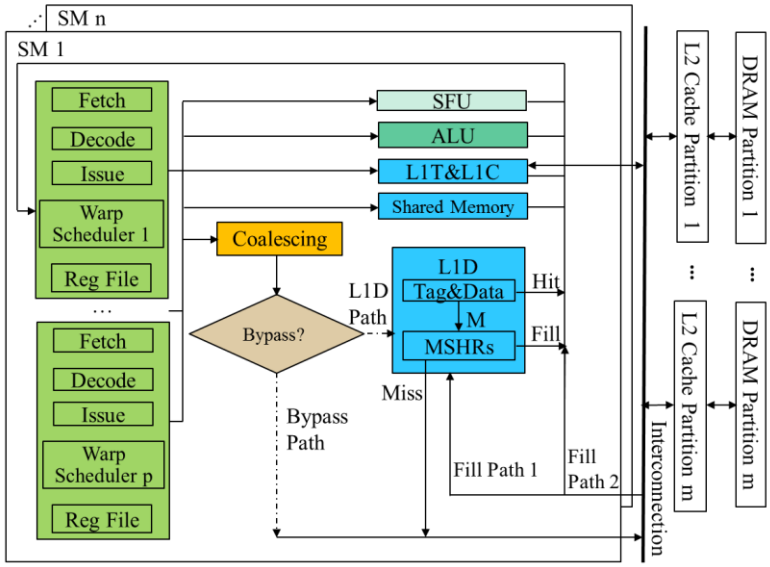


Figure 10. Baseline GPU.

information of bypassed requests and an unlimited number of in-flight bypassed requests can be supported. However, such an assumption is overly optimistic in practice.

MRPB [23] is one of the pioneering works on GPU cache management, inspiring research works on GPU cache bypassing [10][35][36][72] and mitigation of memory pipeline stalls [54][62]. In this work, we investigate how it performs with an altered cache indexing function, cache line allocation policy, MSHR size and memory partition mapping function. In addition to MRPB, we also examine the effectiveness of the GPU cache bypassing scheme MDB[10] with the constraint that only a finite number of in-flight bypassed requests can be supported.

Overall, this work makes the following contributions:

- We show that cache indexing functions have remarkable impact on the overall performance and allocate-on-fill brings significantly higher performance for GPGPU applications, than allocate-on-miss;
- We demonstrate that while more MSHRs can provide a higher MLP, performance is not necessarily improved due to the impact on L1 D-cache performance;

- We present that a well-performing memory partition mapping function should be adopted to achieve more balanced request distribution among memory partitions;
- We illustrate that the effectiveness of prior schemes is reduced with the enhanced baseline and the limitation on the number of in-flight bypassed requests imposes non-trivial impact on GPU cache bypassing schemes.

3.2 Background

As shown in ***Figure 10***, multiple warp schedulers can reside on each SM of a GPU and each scheduler supervises multiple warps. And on each SM, there are on-chip memory resources including a L1 read-only texture cache, a L1 read-only constant cache, a L1 data cache (D-cache), and shared memory. A unified L2 cache is shared among multiple SMs. Typically, the L1 D-cache uses write-evict with either write-allocate [1] or write-no-allocate [43][42] policies, and the L2 cache uses the write-back write-allocate to save the NoC and DRAM bandwidth [57]. Requests sent to the lower level memory hierarchies (L2 cache and DRAM) are distributed among memory partitions based on the memory partition mapping function.

On GPUs, global and local memory requests from threads in a warp are coalesced into as few transactions as possible before being sent to the memory hierarchy. The cached or bypassed information is typically encoded in instruction opcodes [45], indicating whether a request is sent to the L1 D-cache through the 'L1D path' or directly to the L2 cache through the 'Bypass Path', as shown in ***Figure 10***.

For a request sent to the L1 D-cache, the cache indexing function is applied to determine which set to search for the required data and to insert/evict a cache line if it is a miss. Thus, the cache indexing function is crucial to balance requests among cache sets.

Then, for a cache miss, the cache-miss-related resources are allocated before sending the miss request to the L2 cache. For allocate-on-miss, the allocated resources include a cache line slot, a MSHR and a miss queue entry while for allocate-on-fill, just a MSHR and a miss queue entry need to be allocated. If any of the required resources is not available, a reservation failure occurs and the memory pipeline is stalled. Since a MSHR entry is reserved until the

Table 3. Baseline architecture configuration

# of SMs	16, SIMD width=32, 1.4GHz
Per-SM warp schedulers	4 Greedy-Then-Oldest schedulers
Per-SM limit	3072 threads, 96 warps, 16 thread blocks, 64 MSHRs
Per-SM L1D-cache	16KB, 128B line, 4-way associativity
Per-SM shared memory	96KB, 32 banks
Unified L2 cache	2048 KB, 128KB/partition, 128B line, 16-way associativity, 128 MSHRs
L1D/L2 policies	alloc-on-miss, LRU, L1D:WEWN, L2: WBWA
Interconnect	16*16 crossbar, 32B flit size, 1.4GHz
DRAM	16 memory channels/partitions, Modulo mapping, FR-FCFS scheduler, 924MHz

data is fetched from lower memory levels, the MSHR size determines how many in-flight outstanding misses can be supported, i.e., the upper bound of memory-level parallelism.

Although a request sent to the L1 D-cache enjoys low access latency if it hits in the L1 D-cache, the massive requests on GPUs easily cause cache thrashing and cache-miss-related resource congestion, degrading the overall performance. GPU cache bypassing has been proposed to effectively mitigate these problems. Similar to the fact that MSHRs are used to record relevant information of outstanding misses, some hardware structure should be used to store information for bypassed requests, such as which threads in a warp ask for the data and the destination register.

3.3 Experimental Methodology

Simulation Environment: we use GPGPUSim V3.2.2 [5], a cycle-accurate GPU microarchitecture simulator, to evaluate different implementation choices. ***Table 3*** shows the

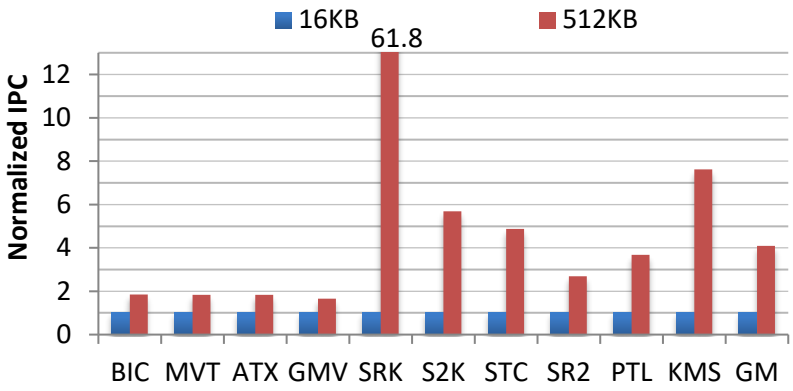


Figure 11. HCC (High Cache Contention) benchmarks from Polybench and Rodinia.

Table 4. Benchmarks

Abbreviation (Description)	Type	Source
BIC (BiCGStab linear solver subkernel)	HCC	[15]
MVT (Matrix-vector-product and transpose)	HCC	[15]
ATX (Matrix-transpose-vector multiply)	HCC	[15]
GMV (Scalar-vector-matrix multiply)	HCC	[15]
SRK (Symmetric rank-2k operations)	HCC	[15]
S2K (Symmetric rank-2k operations)	HCC	[15]
STC (StreamCluster)	HCC	[8]
SRD2 (Srad_v2)	HCC	[8]
PTL (Particle Filter)	HCC	[8]
KMS (K-means clustering)	HCC	[8]

baseline Maxwell-like configuration that has been widely used in GPU architecture studies, including memory architecture studies.

Benchmarks: we evaluate two entire benchmark suites, Rodinia [8] and Polybench [15], including both regular and irregular applications, as listed in ***Table 4***.

Before we start our investigation, we first examine the impact of cache by checking the performance from a small 16KB 4-way set-associative L1 D-cache and a large 512KB full-associative L1 D-cache. Based on the performance improvement from the 512KB L1 D-cache, we classify benchmarks with more than 50% improvement as High Cache Contention (HCC) and others as Low Cache Contention (LCC). Our study focuses on HCC benchmarks as

generally the is not much variance for LCC ones, the same as previous GPU memory architecture studies [7][10][23][36][52][62].

All HCC benchmarks are shown in ***Figure 11*** and ***Table 4***, where performances are normalized to that from the 16KB L1 D-cache and the average performance from the 512 KB L1 D-cache is 4.09x. And the significant performance improvement indicates it is crucial to optimize memory architecture for high performance on GPUs.

3.4 GPU Cache Indexing

In this section, we illustrate the performance impact of cache indexing functions on GPGPU applications and show that a well performing cache indexing function should be deployed in the first place in GPU memory architecture studies.

3.4.1 Performance impact

With limited cache capacity per thread, GPU caches suffer from severe capacity contention. Besides, the capacity may not be well utilized due to conflict misses, which are resulted from column-majored strided accesses in a warp of threads and thus a high number of un-coalesced requests [62]. Furthermore, it has been observed that the baseline Modulo indexing used by default in GPGPUSim may cause pathological performance results [33][36]. Hereby, we thoroughly study the impact of several advanced cache indexing functions to identify the simple and effective one to mitigate conflict misses for GPGPU applications.

As shown in ***Figure 12***, an address is decoded into several fields and the first b bits are the block offset (blk offset), where b = $\log_2$(block size). The next two k-bit fields, where k = $\log_2$(number of sets) in the cache are denoted as x and t.

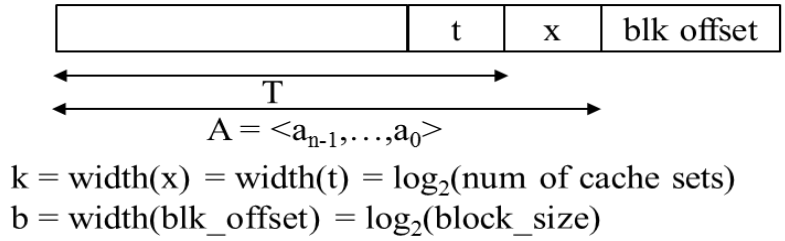


Figure 12. Decoding an address.

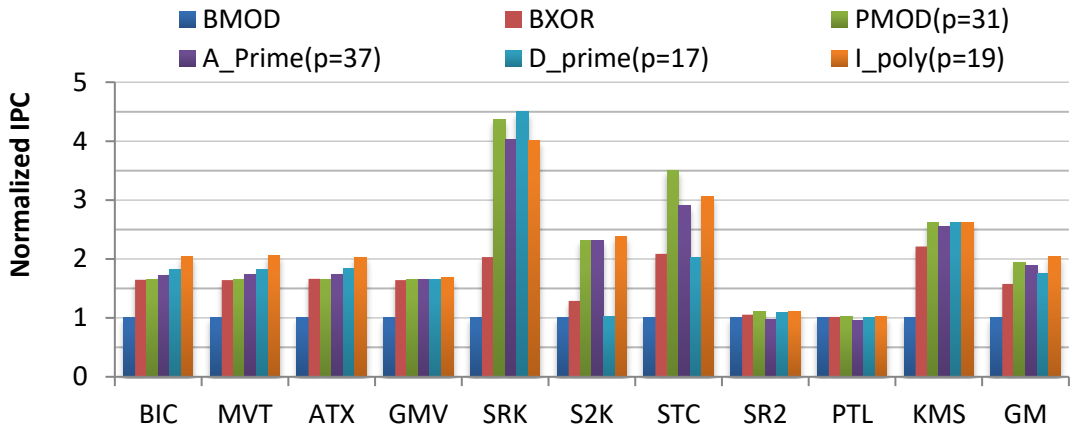


Figure 13. Performance impact of cache indexing functions on HCC benchmarks with baseline cache management.

Baseline Modulo mapping (BMOD): this traditional indexing function defines index as: Index = x = A mod 2^k, where x and A use the same notations as in ***Figure 12***.

Bitwise XOR mapping (BXOR): the bitwise-XOR indexing function [14] defines the index as Index = t^x.

Prime Modulo mapping (PMOD): based on a prime number p, which is closest to but smaller than 2^k, and Index = A mod p [29]. Since this mapping method only uses a prime number of cache sets, it incurs wasted cache sets [38].

Another Prime Modulo mapping (A_Prime): proposed to eliminate the inefficiency of PMOD and Index = A mod p mod 2^k, where p is a prime number closest to but larger than the number of cache sets [38]. In this method, all cache sets are utilized while the first few sets are used more often.

Prime Displacement mapping (D_Prime): In the prime-displacement mapping [29], Index = (T * p + x) mod 2^k, where T and x use the same notations as in ***Figure 12***.

Irreducible Polynomial mapping (I-Poly): The polynomial R(x), determined by R(x) = A(x) mod P(x), is the binary representation of the index and computed as [50]: $R(x) = a_{n-1}R_{n-1}(x) + \ldots + a_1R_1(x) + a_0R_0(x)$, where $R_i(x) = x^i \bmod P(x)$ can be precomputed once P(x) is selected.

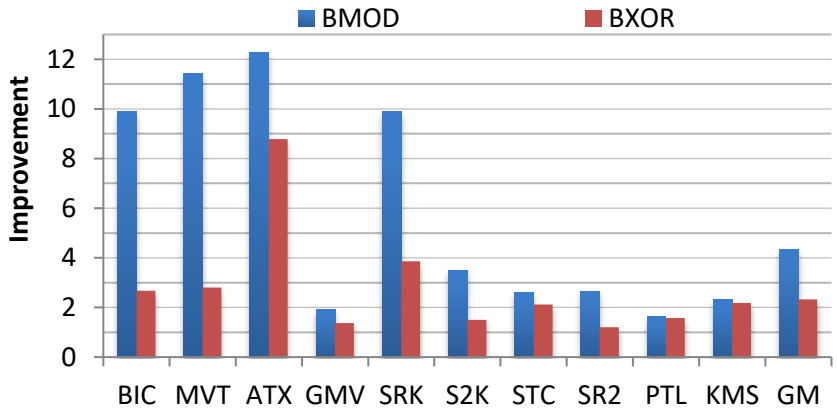


Figure 14. Performance improvement from MRPB with cache indexing functions BMOD and BXOR on HCC benchmarks.

Figure 13 shows performance from different GPU cache indexing functions used upon a 16KB L1 D-cache. First, despite the variations, compared to the BMOD, advanced cache indexing functions result in significantly better performance, which is 1.58x, 1.95x, 1.88x, 1.75x and 2.04x for BXOR, PMOD, A_Prime, D_Prime and I_poly, respectively. Second, while other cache indexing functions either lose their effectiveness in certain cases (like D_prime for S2K and STC) or require more complex computation (like I-poly) which may add latency onto the critical path, BXOR is effective and simple to implement. Therefore, given its cost effectiveness, BXOR is a reasonable choice for cache set indexing. This is consistent with the finding in the work [40], which identified that BXOR is used on commercial GPUs through micro-benchmarking. Third, massive multithreading on GPUs makes their L1 D-caches relatively latency tolerant. Therefore, the more complex indexing functions such as PMOD can lead to significant performance gains. Moreover, the prime-modulo computation can be implemented with simple additions when we use Mersenne primes (i.e., 2p-1).

3.4.2 Effectiveness of MRPB with different cache indexing

On GPU cache management, Jia et al. proposed MRPB [23] which deploys memory request prioritization buffers to reduce the effective working set and bypasses L1 D-cache when a request encounters a reservation failure. It is shown that MRPB significantly improves the performance of HCC benchmarks but it is not clear how cache sets are indexed in their

experiments. Thus, it remains interesting to check the effectiveness of MRPB under both BMOD and BXOR.

As shown in ***Figure 14***, MRPB significantly improves the performance of HCC benchmarks when BMOD is used, matching the experimental results in the work [23]. However, the performance improvement of MRPB over the baseline is greatly reduced when BXOR is adopted. On average, the normalized performance of MRPB over the baseline drops from 4.35x with BMOD to 2.32x with BXOR, confirming that pathological result may occur when BMOD is used.

Given the remarkable performance impact of cache indexing on HCC applications, we believe that a well-performing cache indexing function should be deployed in the first place and BXOR is good to use in terms of cost-effectiveness. The BOXR cache indexing function for both L1D and L2 cache, is used in the subsequent sections.

3.5 Cache Line Allocation

In this section, we dissect the performance impact of the two cache line allocation policies, namely allocate-on-miss and allocate-on-fill.

3.5.1 Performance impact

As described in Section 3.2, when there is an outstanding miss, allocate-on-miss allocates a cache line in addition to a MSHR and a miss queue entry while allocate-on-fill just allocates a MSHR and a miss queue entry. Allocate-on-fill brings in two performance benefits. First, since allocate-on-fill does not evict the victim cache line until the requested data come back from L2 cache/off-chip memory, cache lines have longer lifetime to capture temporal reuses. This is particularly the case for GPUs as the L2 cache latency is much higher than that of CPU L2

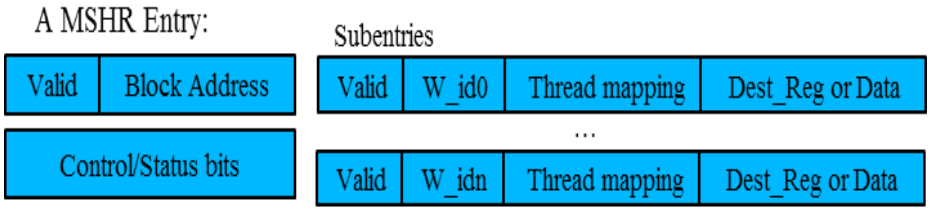


Figure 15. A MSHR Entry on GPUs.

caches since multiple SMs share the L2 cache on a GPU. Besides, the in-order execution model and the write-evict policy make the GPU L1 D-cache friendly to allocate-on-fill as there is no dirty data to write to L2 when a victim cache line is to be evicted at the fill time. Furthermore, as allocate-on-fill does not reserve a cache line slot, cache-miss-related resource congestion is lighter than allocate-on-miss.

Figure 15 shows a MSHR entry on GPUs. The basic structure is similar to the *simple organization* proposed by Tuck et al [60]. The fields of a single MSHR entry include valid bit, block address, control/status bits (prefetch, which subblocks have arrived, etc.). Due to the nature of gather operation on GPUs, thread mapping in subentries tracks which words in the requested cache line map to which threads, as described in WarpPool [30].

Figure 16 (b) shows the performance of allocate-on-miss and allocate-on-fill on various cache capacities, normalized to that of a 16KB L1 D-cache with allocate-on-miss. Individual kernels' performances are also shown for cache capacity 16KB, in ***Figure 16*** (a). As demonstrated, allocate-on-fill consistently outperforms allocate-on-miss. On average, the performance from allocate-on-fill (allocate-on-miss) is 1.4x(1.0x), 5.2x(3.1x) and 10.7x(8.1x) with a 16KB, 32KB and 64KB L1 D-cache, respectively.

The better performance gains of allocate-on-fill comes from a higher L1 D-cache efficiency and relieved cache-miss-related resource congestion. ***Figure 16*** (c) and (d) show L1 D-cache miss rate and L1 D-cache rsfail rate (reservation failures per access), respectively. And take a 32KB L1 D-cache as the example, the L1 D-cache miss rate (rsfail rate) is reduced from 0.48(7.5) with allocate-on-miss to 0.33(2.9) with allocate-on-fill.

3.5.2 Effectiveness of MRPB with different cache line allocation policies

Given the significant performance impact of cache line allocation policies, we also check the effectiveness of MRPB when varying this factor. ***Figure 17***(a) shows the performance from MRPB under both allocate-on-miss and allocate-on-fill for different cache capacities, normalized to the performance from the baseline cache management on a 16KB L1 D-cache with allocate-on-miss. As shown, MRPB remains effective under both cache line allocation policies. And similar to the baseline cache management, allocate-on-fill benefits MRPB by

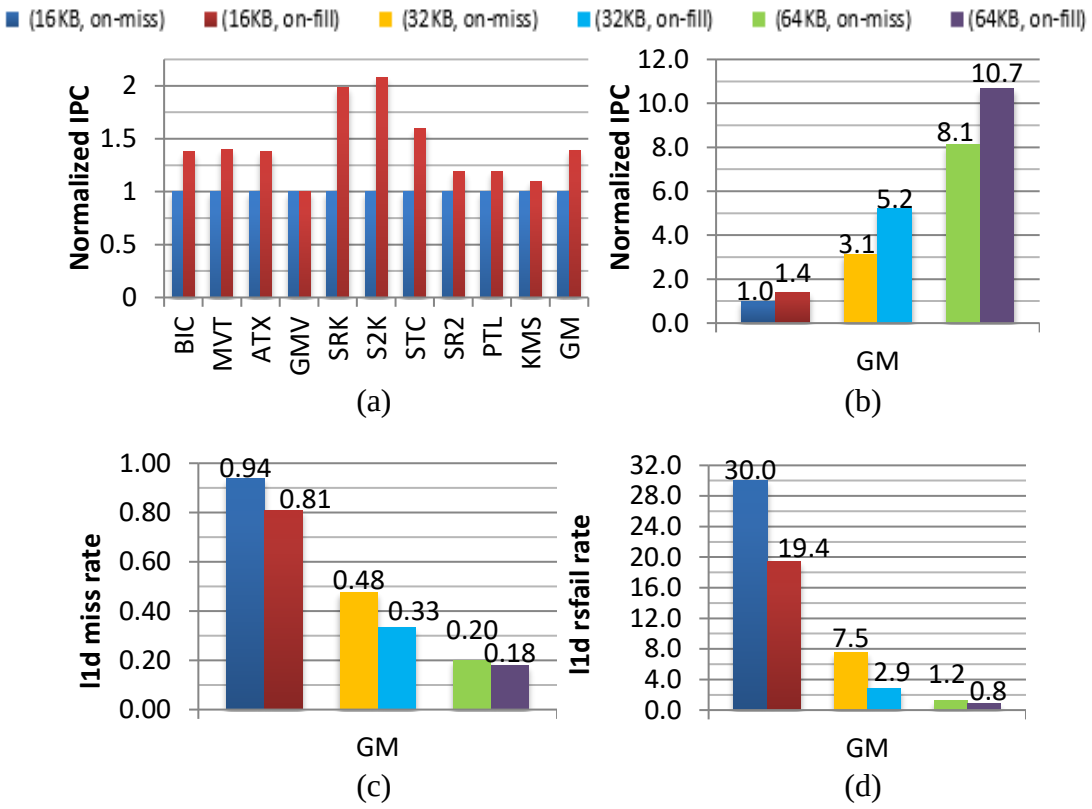


Figure 16. Performance impact of cache line allocation policy with baseline cache management: (a)16KB L1 D-cache: normalized IPC; (b) normalized IPC; (c) L1 D-cache miss rate; (d) L1 D-cache rsfail rate (reservation failures per access).

bringing more hits and fewer reservation failures. Therefore, we can see the performance from allocate-on-fill is higher than that from allocate-on-miss with the same cache capacity. For example, on a 32KB L1 D-cache, the performance increases from 6.4x with allocate-on-miss to 7.2x with allocate-on-fill.

However, since the performance is already boosted for the baseline cache management with allocate-on-fill, the effectiveness of MRPB is reduced. For instance, the performance improvement from MRPB over the baseline on a 32KB L1 D-cache is 110% with allocate-on-miss and it is reduced to 40% with allocate-on-fill, shown in ***Figure 17*** (b).

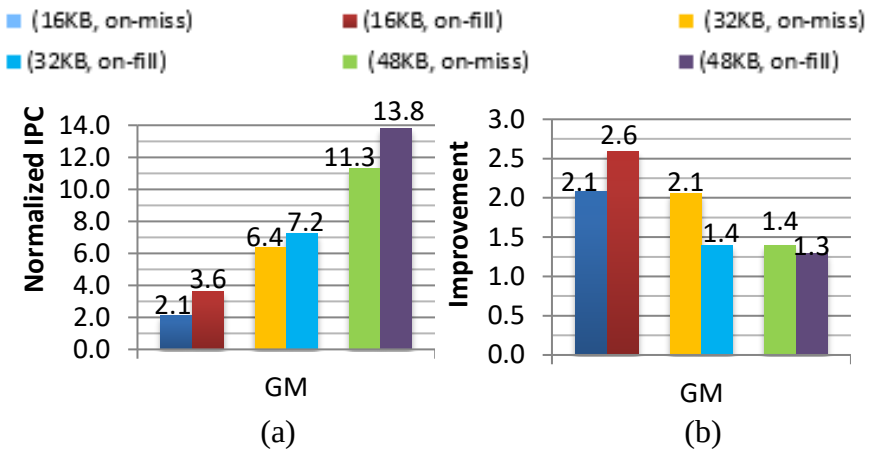


Figure 17. Effectiveness of MRPB with varied cache line allocation policies: (a) normalized IPC; (b) performance improvement over the baseline cache management.

To summarize, considering the non-trivial performance impact, we argue that the two different cache line allocation policies should be taken into account in the evaluation of GPU cache management schemes.

3.6 MSHR Sizes

In this section, we study the impact of the MSHR size. Since a MSHR entry is reserved until the required data is returned from lower memory levels, the number of MSHRs determines the number of outstanding misses which can be served in parallel, i.e., the upper bound of MLP. Besides, a scheduled warp may be stalled in the memory pipeline if all MSHRs are reserved by previous outstanding misses and thus TLP may be reduced with a small number of MSHRs.

3.6.1 Performance impact

First, we show that the MSHR size has a high impact on the overall performance. ***Figure 18***(a) shows the average performance with different MSHR sizes, namely 32, 64, 128 and 256 MSHRs and when the optimal MSHR size is applied to each benchmark, indicated by‘Optimal’. First, while the average performance increases with more MSHRs in most scenarios, an up-then-down performance trend shows up on a 64KB L1 D-cache with allocate-on-miss, indicating more MSHRs do not necessarily bring a better performance, confirming

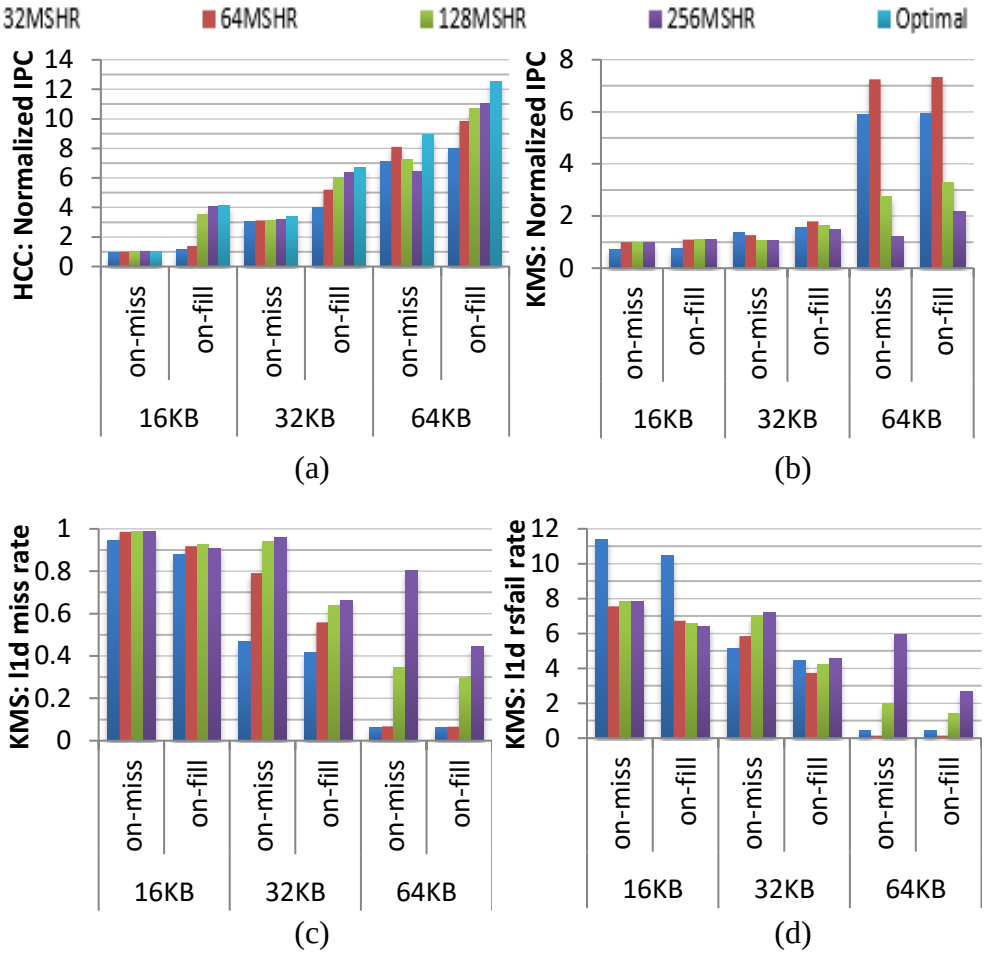


Figure 18. Performance impact of MSHR size: (a) average normalized IPC; (b) KMS: normalized IPC; (c) KMS: L1 D-cache miss rate; (d) KMS: L1 D-cache rsfail rate (reservation failures per access).

the observations in the work [62]. Second, the best performing MSHR size varies for different benchmarks and no single MSHR size can hold the advantage consistently. Thus the 'Optimal' performance can be much better than that from any fixed MSHR size. For example, the normalized IPC of 'Optimal' is 8.94x (12.54x) while the best performance among the fixed MSHR sizes is 8.11x (11.0x) from 64MSHR (256MSHR) for allocate-on-miss (allocate-on-fill) with a 64KB L1 D-cache. Third, as shown, allocate-on-fill is more immune to the potential adverse effect of more MSHRs and thus consistently obtains a better performance with more MSHRs, on average.

3.6.2 The impact of MSHR sizes on L1 D-cache performance

To further investigate the impact of MSHR size, we use KMS as a case study to demonstrate how more MSHRs may hurt the overall performance.

Figure 18(b) presents the performance for KMS from different MSHR sizes, normalized to the performance form a 16 L1 D-cache with allocate-on-miss and 64 MSHRs. For a small 16KB L1 D-cache, where the miss rate remains low, more MSHRs lead to relieved cache-miss-related resource congestion and increased MLP (memory level parallelism), resulting in a better performance. However, for a larger cache, especially, 64KB L1 D-cache, the performance first increases and then decreases with more MSHRs. Specifically, under allocate-on-miss, the performance increases from 5.9x with 32MSHR to 7.3x with 64MSHR because the miss rate remains low (around 0.065) and the number of reservation failures is reduced. Then the performance drops to 2.8x with 128MSHR and to 1.2x with 256MSHR, because both the miss rate and the number of reservation failures significantly increase. And the same variation can also be observed for allocate-on-fill.

To further figure out why the L1 D-cache performance decreases with more MSHRs, we looked into the cycle-by-cycle accesses and found that due to the multithreading execution model of GPUs, in which a new warp is scheduled if the current one is waiting for results of its previous instructions, when there are more MSHRs, more warps are actively scheduled to access L1 D-cache, causing more severe cache thrashing and miss-related resource congestion; on the other hand, with fewer MSHRs, fewer warps can allocate a MSHR and when a request is fulfilled and a reserved MSHR is released, based on the warp scheduling policy, it is highly possible that one of the previously scheduled warps can be scheduled to issue a memory request again and thus has a bigger chance to get a hit in the cache.

To summarize, on GPUs, although more MSHRs can bring a higher MLP, they also enable more requests into caches in a short interval and increase the probability of cache thrashing, confirming the finding in works [40][62] that fewer MSHRs yield memory access throttling and better cache behaviour for some benchmarks. Nevertheless, since 128 MSHRs

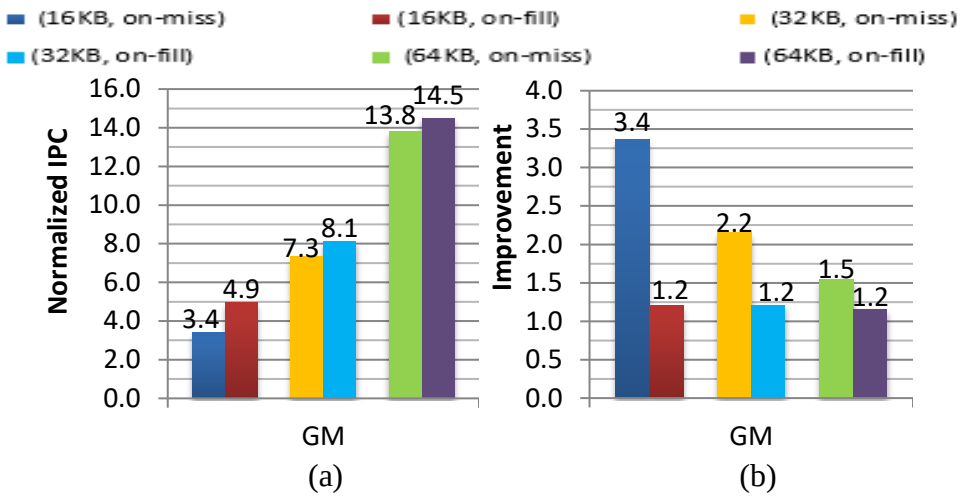


Figure 19. Effectiveness of MRPB with the optimal MSHR size: (a) normalized IPC; (b) performance improvement over the baseline cache management.

can bring a good performance on average in this study, we suggest using 128 MSHRs in the configuration.

3.6.3 Effectiveness of MRPB with the optimal MSHR size

Given the significant performance impact of MSHRs, we also check the effectiveness of MRPB when the optimal MSHR size is applied for each benchmark.

On one hand, the performance of MRPB is further improved with the optimal MSHR size, as shown in ***Figure 19***(a). For instance, the average performance from MRPB is 6.4x (7.2x) on a 32KB L1 D-cache under allocate-on-miss (allocate-on-fill) with 64MSHR (Figure 7(a)) and it is improved to 7.3x (8.1x) when the optimal MSHR size is applied for each benchmark. On the other hand, the performance improvements from MRPB may decrease when the optimal MSHR size is applied to it and the baseline cache management. For example, while the improvement from MRPB over the baseline cache management is 40% on a 32KB L1 D-cache with 64 MSHRs and allocate-on-fill, it is reduced to 20% when considering the optimal MSHR size.

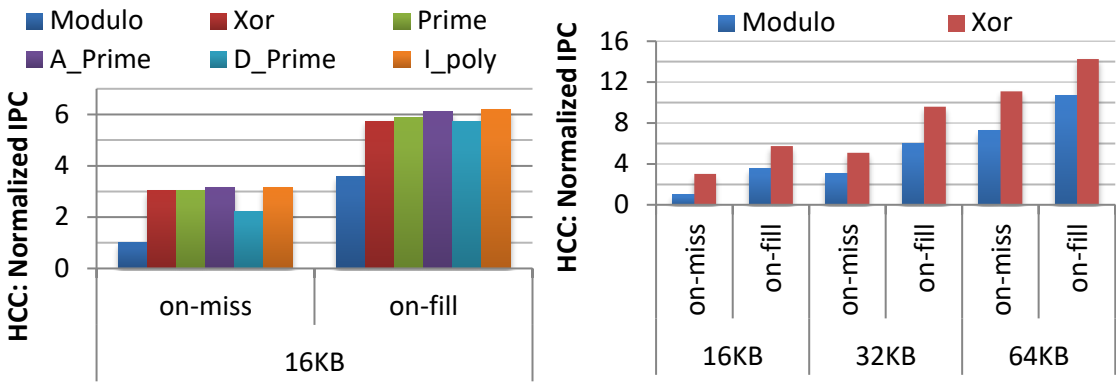


Figure 20. Performance impact of memory partition mapping.

3.7 Request Distribution among Memory Partitions

In this section, we investigate the impact of memory request distribution among partitions. By default in GPGPUsim, addresses are linearly distributed among memory partitions (*Modulo mapping*). However, with *Modulo mapping*, some accesses such as column-majored stride accesses may cause severe partition camping and requests are disproportionately handled by a small subset of memory partitions. Therefore, we thoroughly study the impact of advanced memory partition mapping functions to identify the simple and effective one to mitigate memory partition camping.

Figure 20(a) shows the performance of different memory partition mapping functions used upon a 16KB L1 D-cache, including *Modulo*, Xor, *Prime*(p=13), *A_Prime*(p=31, then modulo 16), D_*Prime*(p=11) and *I-Poly*(p=19), the implementations of which are similar to cache set indexing functions discussed in Section IV. Compared to the baseline Modulo mapping, advanced memory partition mapping functions lead to significantly better performance, which are 3.02x(5.74x), 3.04x(5.87x), 3.17x (6.13x), 2.20x(5.73x) and 3.17x(6.19x) for *Xor*, *Prime*, *A_Prime*, *D_Prime* and *I-Poly* with allocate-on-miss (allocate-on-fill), respectively. Given its cost effectiveness, *Xor mapping* is a reasonable choice. And *A_Prime* with p=31 is also highly promising as 31 is a Mersenne prime and the modulo operation can be implemented with additions.

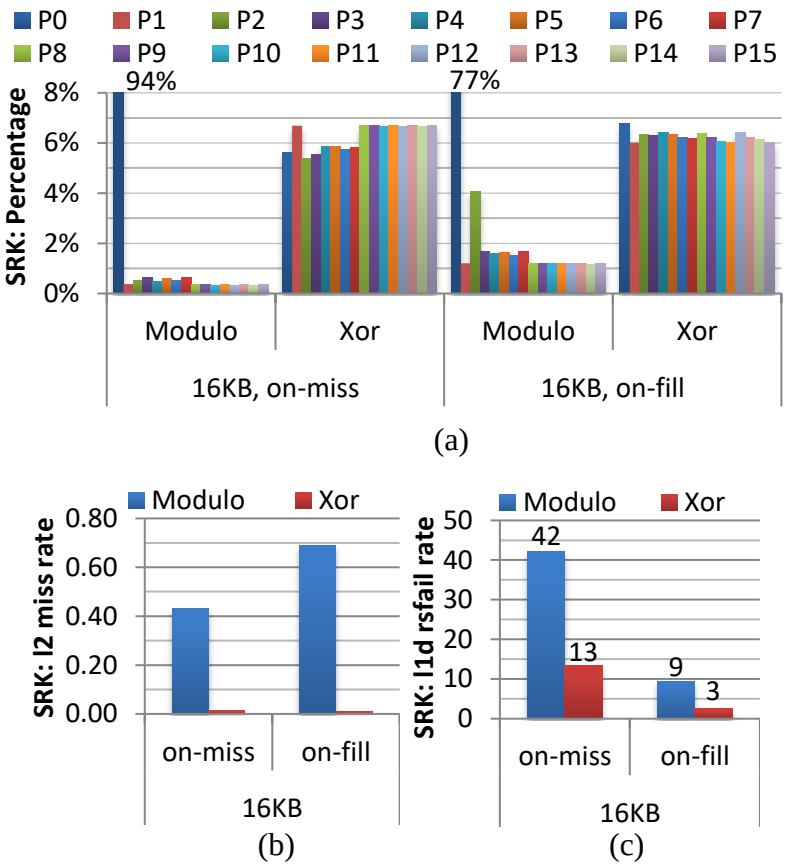


Figure 21. Impact of memory partition mapping: (a) SRK: percentage of requests across 16 memory partitions; (b) SRK: L2 cache miss rate; (c) L1 D-cache rsfail rate (reservation failures per access).

Figure 20(b) shows that *Xor mapping* consistently outperforms *Modulo mapping* across different cache capacities. For example, the normalized IPC is 3.1x (6.0x) for *Modulo mapping* and 5.1x (9.6x) for *Xor mapping* on a 32KB L1 D-cache with allocate-on-miss (allocate-on-fill).

To further investigate how *Xor mapping* outperforms *Modulo mapping*, we use the benchmark SRK for case study and examine memory request distribution among partitions, L2 cache efficiency and L1 D-cache miss-related resources congestion. First, ***Figure 21***(a) shows that a majority of requests are mapped to memory partition 0 with *Modulo mapping*, leading to extremely low DRAM bandwidth utilization. And *Xor mapping* overcomes the problem of memory partition camping by evenly distributing requests among all 16 partitions. The more

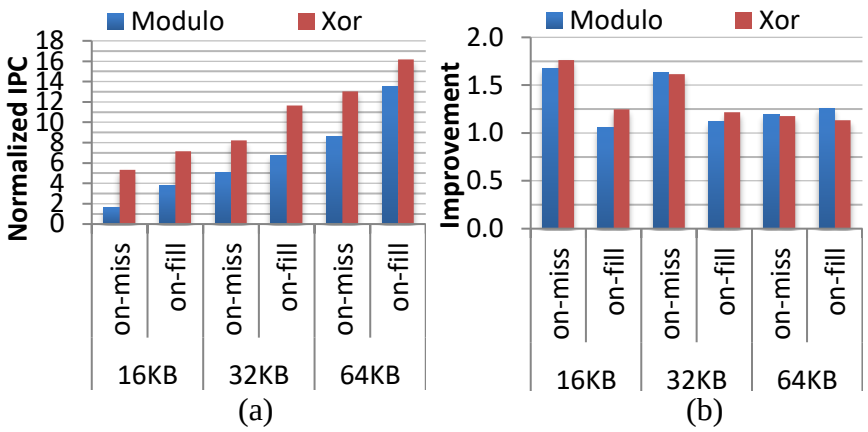


Figure 22. Effectiveness of MRPB with Modulo and Xor memory partition mapping: (a) normalized IPC; (b) improvement over the baseline cache management.

balanced memory request distribution not only greatly improves DRAM bandwidth utilization, but also improves L2 cache efficiency. As shown in ***Figure 21***(b), L2 cache miss rate significantly decreases with *Xor mapping*, due to more balanced accesses to L2 banks and thus better L2 capacity utilization. Furthermore, the improved efficiency/performance at the lower level memory hierarchy also benefits L1 D-cache accesses. ***Figure 21***(c) shows that L1 D-cache rsfail rate (reservation failures per access) drops from 42(9) to 13(2) for allocate-on-miss (allocate-on-fill). This is because almost all requests sent to the lower level memory hierarchy can be absorbed by L2 cache under *Xor mapping* (***Figure 21***(b)). And in turn round trip latency for L1 D-cache misses is significantly reduced, leading to sooner release of MSHRs occupied by those misses and thus relieved congestion in MSHR allocation as well as fewer reservation failures (memory pipeline stalls).

Given the significant performance impact of memory partition mapping, we also check the effectiveness of MRPB when *Xor mapping* is adopted. ***Figure 22***(a) shows that the performance of MRPB is also improved with *Xor mapping*. For example, the normalized IPC increases from 8.6x (13.5x) with *Modulo mapping* to 13.0x (16.2x) with *Xor mapping* on a 64KB L1 D-cache with allocate-on-miss (allocate-on-fill). Regarding the improvement over the baseline (***Figure 22***(b)), it decreases in some scenarios and increases in others since different benchmarks may react differently when the memory partition mapping function

alters. Nevertheless, it is crucial to ensure balanced request distribution among memory partitions and L2 cache banks.

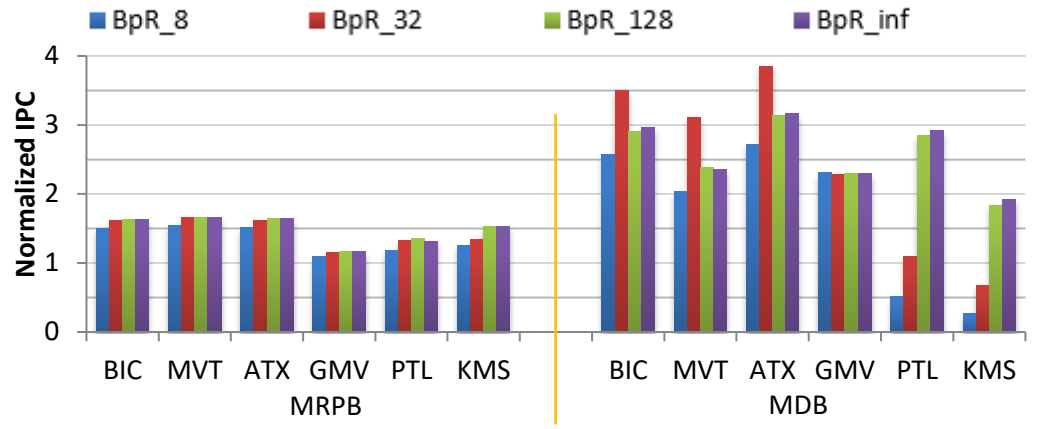
3.8 Modelling Realistic GPU Cache Bypassing

In this section, we investigate how bypassing schemes perform when dedicated hardware structures are allocated to record the relevant information of bypassed requests and thus just a finite number of in-flight bypassed requests can be supported. Works [7][10][23][34][36][72] on GPU cache management have demonstrated that intelligent cache bypassing can significantly improve the overall performance but failed to discuss the constraint from hardware structures used to keep the relevant information of bypassed requests. However, it is unrealistic to assume that an unlimited number of in-flight bypassed requests can be supported.

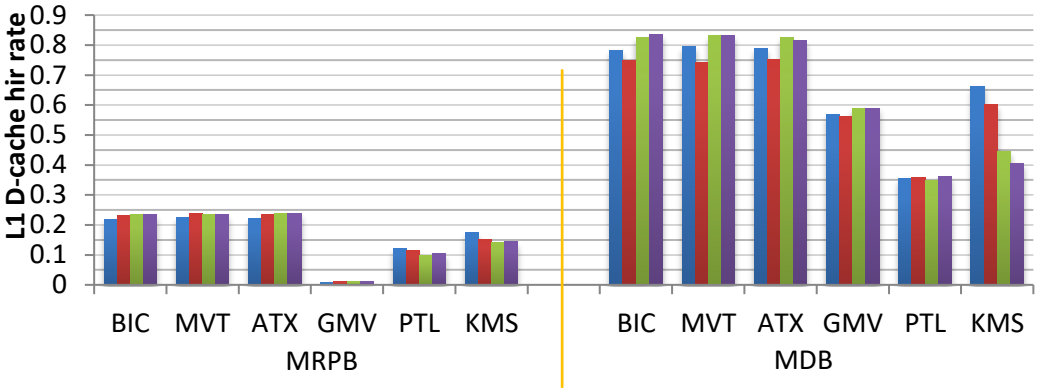
Similar to prior GPU cache bypassing studies, allocate-on-miss is used and 128 MSHRs are deployed for a 16KB L1 D-cache in this study. #_BpR is used to denote the maximum number of in-flight bypassed requests that can be supported. And hardware structures similar to but simpler than regular MSHRs (***Figure 15***) can be used to keep the relevant information of bypassed requests such as which threads ask for the data and the destination register.

Figure 23(a) shows the performance (normalized to the baseline cache management without bypass) of MRPB and MDB [10] when different numbers of in-flight bypassed requests can be served in parallel on a 16 KB L1 D-cache. MDB is a model-drive approach for GPU cache bypassing and it bypasses a certain number of warps or thread blocks based on the combined impact of cache contention and cache-miss-related resource congestion. Here we only show representative benchmarks with diverse and significant performance variance when BpR_# changes. BpR_Inf is used by default in prior GPU cache bypassing works, in which any determined bypassing request can be sent to lower memory levels since there is no limitation from hardware to store the relevant information of bypassed requests.

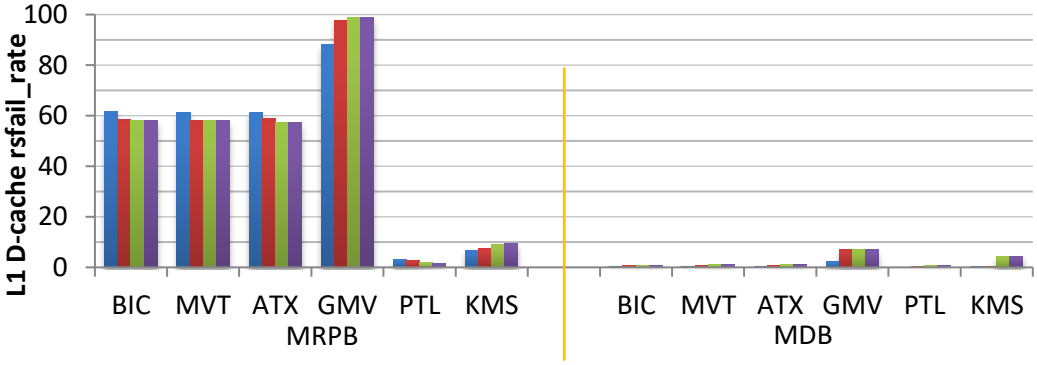
As shown in ***Figure 23***, MDB outperforms MRPB. For instance, when there is no constraint regarding the number of inflight bypassed requests, the performance for BICG, MVT, ATAX, GEMV, PTFL and KMNS from MRPB is 1.63x, 1.66x, 1.64x, 1.16x, 1.32x and 1.53 respectively while it increases to 2.97x, 2.35x, 3.16x, 2.30x, 2.92x and 1.91x from MDB.



(a) Performance



(b) L1 D-cache hit rate



(c) L1 D-cache rsfail_rate

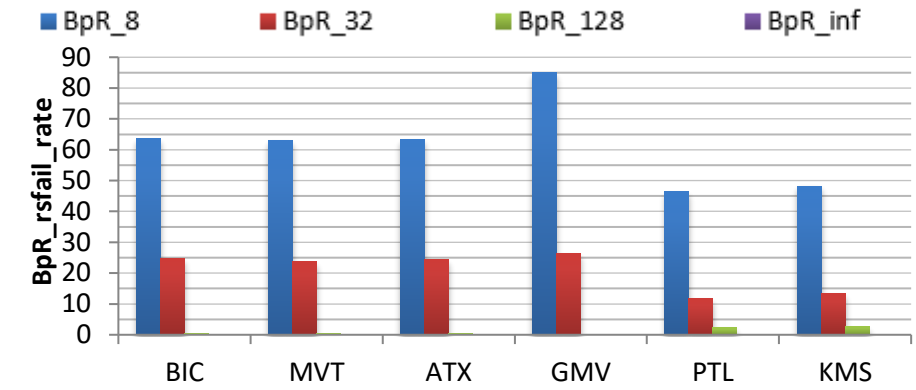
Figure 23. Effectiveness of MRPB and MDB when different numbers of in-flight bypassed requests can be supported.

And MDB constantly performs better than MRPB when various limited number of inflight bypassed requests are supported. Two L1 D-cache metrics, L1 D-cache hit rate and L1 D-cache rsfail_rate (reservation failure per request due to cache-miss-related resource congestion), are

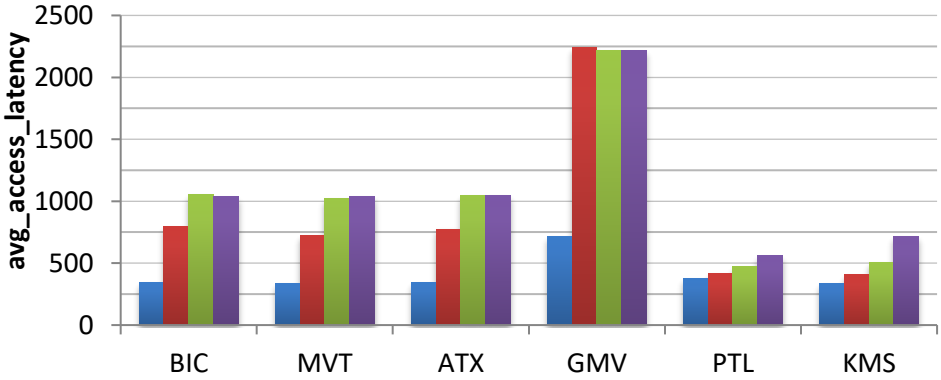
shown in ***Figure 23***(b) and (c) to help interpret the performance difference between MRPB and MDB. First, MDB boosts L1 D-cache hit rate more effectively than MRPB. For example, L1 D-cache hit rate for BICG and KMNS is 0.24 and 0.14 from MRPB, while it increases to 0.83 and 0.41 from MDB. Second, MDB also significantly reduces cache-miss-related resource congestion. For instance, L1 D-cache rsfail-rate for BICG and PTFL is 57.9 and 9.3 from MRPB, while it decreases to 0.9 and 0.7 from MDB. Therefore, the highly improved L1 D-cache efficiency explains the better performance of MDB than MRPB. Since benchmarks show more significant performance improvement and also more observable diversity with different values of BpR_# with MDB, we use MDB to further investigate the impact of BpR_# in the following discussion.

It is intuitive to think that the higher BpR_#, the better the performance since the constraint from such a factor is relieved. However, it is not always the case and the examined benchmarks may show diverse behaviors, as shown in ***Figure 23***(a). First, as expected, the performance increases from BpR_8 to BpR_32 for most of the examined benchmarks, except GMV. Then, from BpR_32 to BpR_128, there is significant performance degradation for benchmarks BIC, MVT and ATX while the performance of benchmarks GMV remains relatively stables and benchmarks PTL and KMS continue to obtain performance improvement. Finally, there is not much variation between BpR_128 and BpR_inf across all the examined benchmarks.

To better understand the impact of BpR_#, we further studied the following two metrics: L1 D-cache BpR_rsfail_rate and average memory access latency. L1 D-cache BpR_rsfail_rate denotes the number of Reservation Failures per bypassed request due to the constraint from BpR_# and such a reservation failure occurs when a new request is determined to bypass the L1 D-cache but the BpR_# has already been reached by prior bypassed requests. The metric, average memory access latency, represents the time interval between when a m request is sent to the memory hierarchy and when the required data comes back to the requesting SM.



(a) L1 D-cache BpR_rsfail_rate



(b) Average memory access latency

Figure 24. Reservation failures due to the constraint on the number of inflight bypassed requests and average access latency in MDB when different numbers of in-flight bypassed requests can be supported.

Figure 24(a) shows L1 D-cache BpR_rsfail_rate with BpR_8, BpR_32, BpR_128 and BpR_inf. And we have the following observations. First, with BpR_8 where 8 in-flight bypassed requests can be supported in maximum, there are a large number of reservation failures due to the constraint of BpR_# and in turn unsuccessful bypass attempts, resulting in severe memory pipeline stalls. In other words, the effectiveness of GPU cache bypassing may be undermined if just a small number of in-flight bypassed request can be supported. Second, L1 D-cache BpR_rsfail_rate significantly drops from BpR_8 to BpR_32 and this leads to the performance improvement from BpR_8 to BpR_32. Third, L1 D-cache BpR_rsfail_rate continues to drop from BpR_32 to BpR_128 and BpR_inf and there is almost no reservation failures due to the limitation of BpR_# for BpR_128 and BpR_inf.

However, although L1 D-cache BpR_rsfail_rate is near-zero for BpR_128 and BpR_inf, there performance is not necessarily better compared to that when fewer inflight bypassed requests can be supported. For instance, the normalized IPC drops from 3.49x, 3.11x and 3.85x with BpR_32 to 2.90x, 2.39x and 3.13x with BpR_128 for benchmarks BIC, MVT and ATX, respectively, as shown in ***Figure 23***(a). Such performance degradation occurs due to the lengthened memory access latency, as shown in ***Figure 24***(b). Specifically, the average memory access latency increases from 793, 727 and 774 cycles with BpR_32 to 1051, 1019 and 1047 cycles with BpR_128 for BIC, MVT and ATX.

Despite that benchmarks BIC, MVT and ATX show performance degradation from BpR_32 to BpR_128, benchmarks PTL and KMS obtain continuous performance improvement with a larger BpR_#. Similar to other benchmarks, PTFL and KMNS encounter fewer reservation failures and lengthened memory access latency when more inflight bypassed requests are supported. The difference between these two and other benchmarks lies in the values of memory access latency with a larger BpR_#. For instance, while the average memory access latency increases from 793 cycles with BpR_32 to 1051 cycles with BpR_128 for BIC, it just increases from 413 to 469 for PTL and from 405 to 507 for KMS. Thus even though the average memory access latency is lengthened for PTL and KMS, it still has a relatively low value and does not offset the benefits brought by fewer reservation failures due to the constraint of BpR_#. In contrast, since the average memory access latency of GMV is more than 2200 cycles starting from BpR_32 and the benefits from fewer reservation failures are offset, the performance of GMV remains relatively stable across all examined values of BpR_#.

A request, which encounters a miss at L1 D-cache, goes through the interconnect network and then gets served by either L2 cache or DRAM. Therefore, the access latency of such a request has two major parts, one is to go through the interconnect network and the other is to be accommodated by L2 cache or DRAM.

Thus to further investigate the impact of BpR_#, we check three more metrics, L2 cache miss rate, avg_BpR_in_circle and max_BpR_in_circle, the latter two of which denote the average/max number of inflight bypassed requests during execution, as shown in Figure 25.

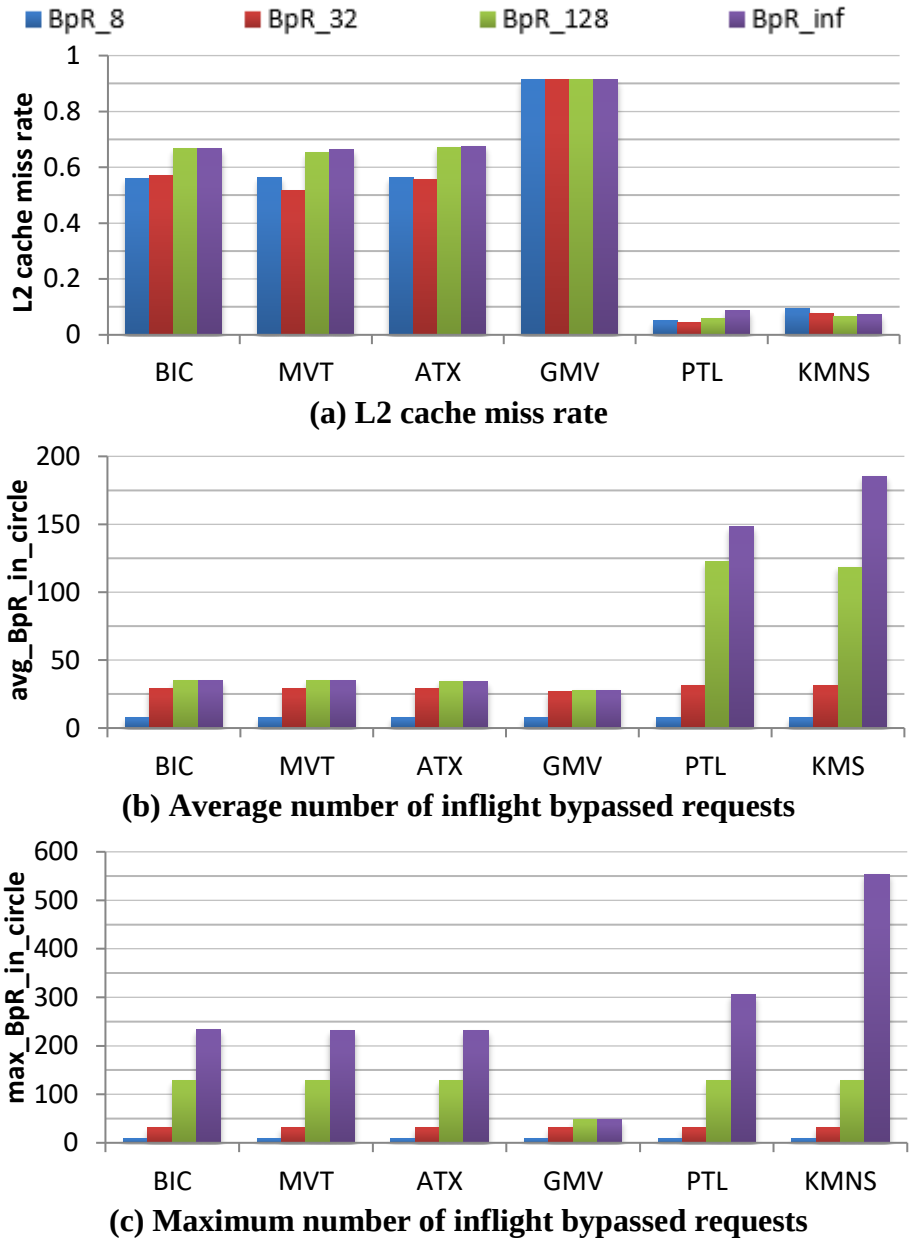
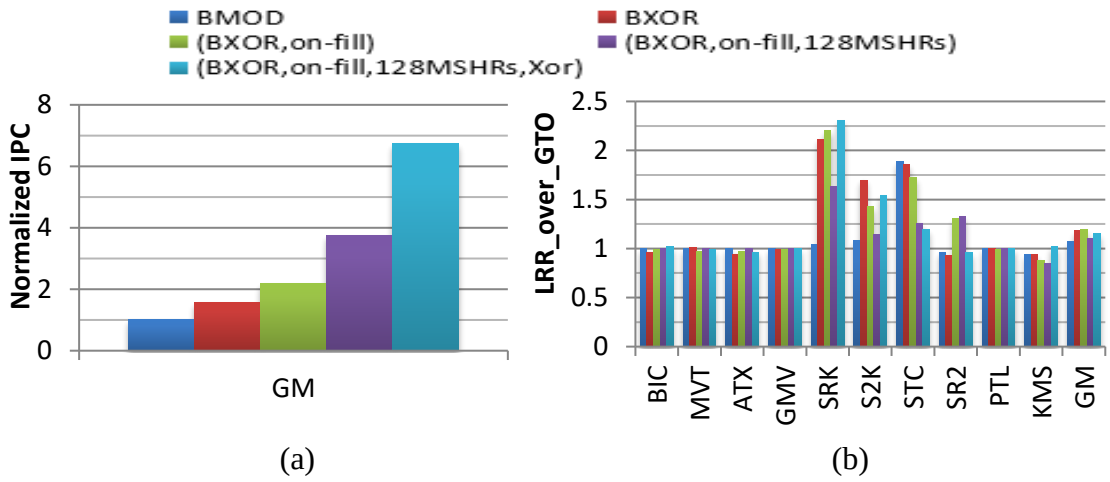


Figure 25. L2 cache efficiency and the number of inflight bypassed requests in MDB when different numbers of in-flight bypassed requests can be supported.

Among the three metrics, L2 cache miss rate indicates the L2 cache efficiency and the higher L2 cache miss rate, the more requests are sent to DRAM and the longer latency for a request to be served by L2 cache and DRAM. The other two metrics avg_BpR_in_circle and max_BpR_in_circle reflect the extent of interconnect congestion. The larger values of

avg_BpR_in_circle and max_BpR_in_circle, the higher degree of interconnect congestion and the longer latency for a L1 D-cache miss goes through it. First, for benchmarks BIC, MVT and ATX, there is non-trivia L2 cache miss rate increase from BpR_32 to BpR_128 because more warps are actively scheduled to send requests to the memory subsystem, implying a longer latency for a request to be served by L2 cache or DRAM. In the meanwhile, since there are more inflight bypassed requests, as shown in ***Figure 25***(b) and (c), the latency to go through the interconnect network also increases. The combined effect of the two factors leads to the significantly lengthened memory access latency and performance degradation for the three benchmarks. For the benchmark PTL and KMS, L2 cache miss rate remains lower than 0.1 for various BpR_#, indicating that more inflight bypassed requests do not thrash L2 cache and almost all L1 D-cache misses can be absorbed by it. On the other hand, unlike other benchmarks, PTL and KMS have a large number of bypassed requests. For example, as shown in Figure 25 (c), with BpR_128, the value of avg_BpR_in_circle for BIC is 35, and it is as high as 123 for PTL and 118 for KMS. Since PTL and KMS experience a much shorter memory access latency and have more inflight bypassed requests, they can have more requests served and in turn execute more data-dependent computation instructions per cycle on average compared to other benchmarks, and therefore obtains continuous performance improvement with a larger BpR_#.

As demonstrated, the number of in-flight bypassed requests can significantly affect the performance of GPU cache bypassing schemes. Therefore, it is not realistic to assume there are unlimited hardware resources to store the relevant information of bypassed requests. On the other hand, the higher number of in-flight bypassed requests to be supported does not necessarily brings higher performance due to the congestion interconnect network and conflicts at lower memory levels. Besides, a limitation on the number of in-flight bypassed requests can also achieve bypass throttling, which is targeted by some prior works [7][36]. So, we believe the fact that only a limited number of in-flight bypassed requests can be supported should be taken into account, to get more accurate results and conclusions in GPU cache bypassing studies.



(a) (b)

Figure 26. With an enhanced baseline: (a) accumulated performance improvement with a 16KB L1 D-cache; (b) relative performance of LRR over GTO.

3.9 Concluded Sound Baseline GPU Cache Design

In this part, we give out the suggested sound baseline GPU cache design. Based on our study, we argue for the following methodology to be used in GPU memory architecture research: (1) An advanced indexing function such as BXOR to reduce conflict misses in the caches; (2) Allocation-on-fill policy in the GPU caches to improve the cache utilization; (3) For studies on memory-level parallelism, the number of MSHRs needs to be explored as an important design space parameter; (4) Studies on cache bypassing should not assume unlimited number of bypasses. Instead, the bypass slots (i.e., the maximal number of in-flight bypasses) is an important design space parameter to be explored.

If a single baseline is desired (i.e., no design space exploration), the sound one from our results is:

BXOR + allocate-on-fill + 128 MSHRs + 32/128 bypassing slots, with *Xor mapping* used to distribute requests among memory partitions.

The sound baseline is open sourced at:
https://github.com/ShadowArray/WDDD-Sound-Baseline

Regarding the performance of the enhanced baseline, we show the accumulated performance improvement with a 16KB L1 D-cache, in ***Figure 26***(a). Without specific description in the legend, the default configuration is (BMOD, on-miss, 64MSHRs, Modulo) which uses BMOD for cache set indexing and allocate-on-miss as the cache line allocation policy, deploys 64MSHRs and distributes requests among memory partitions with Modulo mapping. As shown, the performance continuously increases when the baseline is enhanced. And on average, the accumulated performance is as high as 6.7x with (BXOR, on-fill, 128MSHRs, Xor), compared to the default configuration in GPGPUsim. Although not shown, the enhanced baseline can also benefit LCC (Low Cache Contention) benchmarks. For the indexing function, it may not be the best performance-wise as some other hashing functions may distribute the accesses more evenly than BXOR. But considering the hardware complexity, our results suggest that BXOR is the best one for most applications. For allocation-on-fill vs. allocation-on-miss, allocation-on-fill extends the life-time of the cached data. Therefore, it is better than allocation-on-miss in general. And 128 MSHRs can greatly mitigate reservation failures in this part. Finally, as illustrated in Section 3.7, applications show diverse behaviors when more inflight bypassed requests can be supported. Some applications show an up-then-down performance trend, some achieve a stable performance after a certain point and others can continuously reap performance improvement. As such, we still suggest two points, 32 and 128 bypassing slots should be studied. On one hand, the configuration of 32 bypassing slots can achieve bypass throttling which is target by some prior works [7][36]. On the other hand, as shown in our experiments, 128 bypassing slots can achieve performance close to that when there is no constraint on the number of inflight bypassed requests and since it leads to a higher memory-level parallelism, potentially it can benefit applications for which L2 cache has a high efficiency and can effectively filter requests sent to it.

In addition to Greedy-Then-Oldest (GTO) used so far, we have also experimented Loose-Round-Robin (LRR) warp scheduling policy and found that the overall performance is also boosted with the enhanced baseline as the memory subsystem efficiency is improved.

As demonstrated by prior works [39][52], the warp scheduling policy can have significant performance impact for GPGPU applications and workload classification can be done based on which policy performs better. However, ***Figure 26***(b) shows that the relative advantage may alter with an enhanced baseline. Specifically, for SRK and S2K, the performance is similar with the default configuration and then LRR performs better; for STC, the performance gap continuously shrinks, from 1.9x to 1.2x; for KMS, GTO is better except for the final configuration where LRR and GTO show similar performance. Therefore, an enhanced baseline is also necessary for studies on GPU warp scheduling policies for more realistic results/conclusions.

3.10 Related Work

Although cache indexing functions have been well studied on CPUs [29][14][38], previous works on GPU cache management did not elaborate on this issue in detail. On one hand, some works did not mention the adopted cache indexing function, like MRPB [23]. WarpPool [30] and so on. On the other hand, although some other works pointed out that the simple BMOD mapping used in GPGPUSim is not realistic and might cause pathological results [33][36][40], they did not thoroughly study the impact of various advanced cache indexing functions.

Cache line allocation policy determines what cache-miss-related resources are allocated for an outstanding miss before sending the missing request to lower memory levels. For allocate-on-miss, those resources include a cache line [31], a MSHR and miss queue entry while allocate-on-fill [5] does not reserve a cache line. Therefore, allocate-on-fill tends to incur fewer reservation failures and more hits, leading to a better performance. Besides, although some research works [40][54] have mentioned the potential performance impact of MSHR size on GPUs, they did not study it in detail nor examine the impact with varying other factors. In contrast, we studied the impact of MSHR size with different cache sizes, cache line allocation policies and so on.

Although many prior works have studied GPU cache bypassing and shown significant performance improvements from their schemes [7][10][23][34][36][72], they did not mention the constraint from the hardware structures used to store the relevant information of bypassed

requests. Since only a limited number of in-flight bypassed requests can be supported in reality, we demonstrate that it should be taken into account in GPU cache bypassing studies.

3.11 Conclusions

As throughput oriented processors, GPUs leverage massive multithreading to hide long operation latencies. However, the massive memory requests in GPGPU applications lead to fewer cache lines per thread and shorter cache line lifetime on GPUs than CPUs. In this work, we comprehensively investigated the performance impact of cache set indexing, cache line allocation policy, the number of MSHRs, and request distribution among memory partitions on GPUs as well as more realistic GPU cache bypassing.

Our studies show that advanced cache indexing functions should be deployed in the first place to reduce the severe conflict misses; allocate-on-fill should be used to increase cache hits and reduce memory pipeline stalls; the number of MSHRs plays an important role in affecting the cache efficiency besides supporting MLP/TLP. Furthermore, we show that a good memory partition mapping function, such as Xor, should be deployed to mitigate the problem of memory camping. And while previous GPU cache bypassing works unrealistically assume an unlimited number of in-flight bypassed requests can be supported, we demonstrate such a constraint can significantly affect the performance of a GPU cache bypassing scheme and this factor should be taken into account in GPU cache bypassing studies. Finally, we propose the sound baseline configuration for future GPU memory architecture studies and open source it.

Chapter 4

Accelerate GPU Concurrent Kernel Execution by Mitigating Memory Pipeline Stalls

4.1 Introduction

Following the technology scaling trend, modern GPUs integrate an increasing amount of computing resources [1][67][68][69][70]. Since GPUs have become prevalent in high performance computing, they need to support applications with diverse resource requirements. As a result, GPU resources are typically underutilized by a single kernel.

To solve the problem of GPU resource underutilization, concurrent kernel execution (CKE) [27] has been proposed to support running multiple kernels concurrently on a GPU. One approach to achieve concurrent kernel execution is to apply the left-over policy, in which resources are assigned to one kernel as much as possible and the leftover resources are then used for another kernel. The examples implementing this approach include the queue-based multiprogramming [51][53] introduced by AMD and Hyper-Q by NVIDIA [67]. However, the simple left-over policy fails to optimize resource utilization and does not provide fairness or quality of service (QoS) to concurrent kernels.

Researchers have proposed software and hardware schemes to better exploit CKE. Studies have shown that CKE improves GPU resource utilization especially when kernels with complementary characteristics are running together. Models [24] [37] [75] aim to find the optimal pair of kernels to run concurrently. Kernel slicing [75] partitions a big kernel into smaller ones such that no single kernel consumes all resources. Elastic kernel [48] dynamically adjusts the kernel size based on the resource availability. Those approaches have demonstrated the advantages of CKE. However, it may not be feasible to modify every application. To exploit CKE more broadly, we focus on hardware approaches in this work.

One hardware-based concurrent kernel execution scheme is spatial multitasking [2], which groups streaming multiprocessors (SMs) in a GPU into multiple sets and each set can execute a different kernel. Such SM partition enables better fairness among concurrent kernels but does not overcome resource underutilization within an SM. For instance, the computing resources in an SM, which runs a memory-intensive kernel, cannot be utilized for a compute-intensive kernel running on other SMs.

One appealing approach to improve resource utilization within an SM is intra-SM sharing, in which thread blocks from different kernels can be dispatched to one SM. The intuitive idea is to run kernels with complementary characteristics concurrently on an SM, such as a compute-intensive kernel and a memory-intensive one.

SMK [65] and Warped-Slicer [71] are two state-of-the-art intra-SM sharing schemes, and they adopt different algorithms to determine Thread-Block (TB) partition among concurrent kernels, i.e. how many TBs can be issued from individual kernels to the same SM. Specifically, SMK uses the metric 'Dominant Resource Fairness (DRF)' to fairly allocate static resources (including registers, shared memory, number of active threads and number of TBs) among concurrent kernels. Since good fairness in static resource allocation does not necessarily lead to good performance fairness, SMK also periodically allocates quotas of warp instructions for individual kernels based on profiling each kernel in isolation. On the other hand, Warped-Slicer determines TB partition based on scalability curves (performance vs. the number of TBs from a kernel in an SM) which are also generated by profiling each kernel in isolation. The TB partition with which the performance degradation of individual kernels is minimized is identified as the sweet point.

Although SMK and Warped-Slicer outperform spatial multitasking, they both rely on profiling individual kernels in isolation and fail to address the interference among concurrent kernels which may hurt the overall performance. First, since kernels share the same memory pipeline in intra-SM sharing, one kernel will starve if it constantly loses competition to issue memory instructions. Specifically, as compute-intensive kernels have significantly fewer memory instructions than memory-intensive ones, their memory instructions tend to be

delayed, leading to their severe performance loss. Second, as shown in previous works [10] [54], memory pipeline stalls caused by cache-miss-related resource saturation prevent ready warps from issuing new memory instructions. In intra-SM sharing, such memory pipeline stalls and L1 D-cache thrashing caused by one kernel will impose stalls on all other co-running kernels, resulting in performance degradation.

To overcome the aforementioned issues in intra-SM sharing on GPUs, this paper explores the following approaches. First, we investigate the effectiveness of cache partitioning and highlight that cache partitioning cannot effectively reduce memory pipeline stalls. Second, we propose to balance memory requests such that a compute-intensive kernel does not undergo starvation in accessing the shared memory subsystem. Third, we propose memory instruction limiting to control the number of inflight memory instructions from individual kernels so as to reduce memory pipeline stall and relieve L1 D-cache thrashing. With the proposed schemes, we can reduce interference among concurrent kernels and mitigate memory pipeline stalls, thereby achieving higher computing resource utilization, throughput and fairness.

Overall, this paper makes the following contributions:

- We demonstrate that while the state-of-art intra-SM sharing schemes can identify a good performing TB partition, they profile each kernel in isolation and fail to address the interference among concurrent kernels.
- We show that while cache partitioning cannot reduce memory pipeline stalls, it is beneficial to balance memory accesses and limit the number of inflight memory instructions from concurrent kernels;
- Our experiments show that compared to the two state-of-art intra-SM sharing schemes: our approaches improve the system throughput by 26.5%/27.2% and fairness by 44.2%/85.9% over Warped-Slicer/SMK.

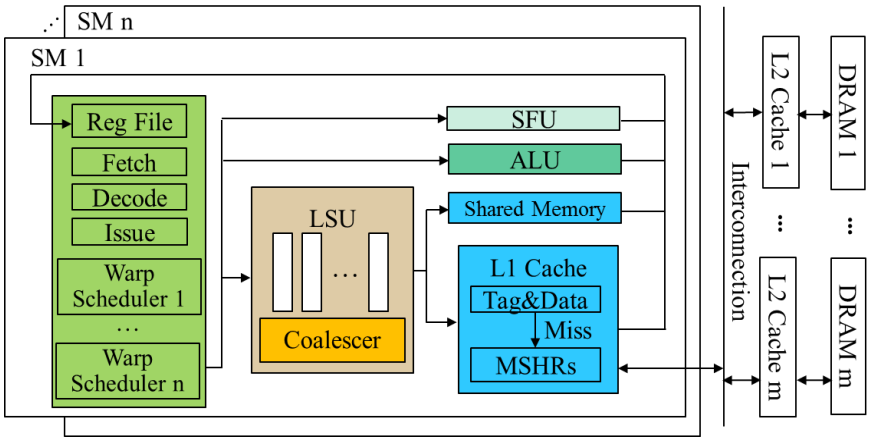


Figure 27. Baseline GPU.

4.2 Motivation and Methodology

4.2.1 Baseline Architecture and Memory Request Handling

As shown in ***Figure 27***, modern GPUs consist of multiple streaming multiprocessors (SMs). A GPU kernel is launched with a grid of thread blocks (TBs). Threads within a TB form multiple warps and all threads in a warp execute instructions in a SIMD manner. More than one warp scheduler can reside in one SM.

Besides massive multithreading, GPUs have adopted multi-level cache hierarchies to mitigate long off-chip memory access latencies. Within each SM, the on-chip memory resources include a read-only texture cache and a constant cache, an L1 data cache (D-cache), and shared memory. A unified L2 cache is shared among multiple SMs.

On GPUs, global and local memory requests from threads in a warp are coalesced into as few transactions as possible before being sent to the memory hierarchy. For a request sent to the L1 D-cache, if it is a hit, the required data is returned immediately; if it is a miss, cache-miss-related resources are allocated, including a miss status handling register (MSHR) and a miss queue entry, and then the request is sent to the L2 cache. If any of the required resources is not available, a reservation failure occurs and the memory pipeline is stalled. The allocated

Table 5. Baseline architecture configuration

# of SMs	16, SIMD width=32, 1.4GHz
Per-SM warp schedulers	4 Greedy-Then-Oldest schedulers
Per-SM limit	3072 threads, 96 warps, 16 thread blocks, 128 MSHRs
Per-SM L1D-cache	24KB, 128B line, 6-way associativity
Per-SM SMEM	96KB, 32 banks
Unified L2 cache	2048 KB, 128KB/partition, 128B line, 16-way associativity, 128 MSHRs
L1D/L2 policies	xor-indexing, allocate-on-miss, LRU, L1D:WEWN, L2: WBWA
Interconnect	16*16 crossbar, 32B flit size, 1.4GHz
DRAM	16 memory channels, FR-FCFS scheduler, 924MHz, BW: 48bytes/cycle

MSHR is reserved until the data is fetched from the L2 cache/off-chip memory while the miss queue entry is released once the miss request is sent to the L2 cache.

4.2.2 Multiprogramming Support in GPUs

Since GPUs continue incorporating an increasing amount of computing resources, CKE is introduced to improve resource utilization. With the Hyper-Q architecture [67], kernels are mapped into multiple stream queues. Grid launch inside a GPU kernel has been proposed to reduce costly CPU intervention [26]. The HSA foundation [51] introduced a queue based multi-programming approach for heterogeneous systems with GPUs. However, no publicly available documentation describes how memory instructions are selected to issue from individual kernels when they are running concurrently in the same SM. Thus, it is crucial to understand the management schemes of memory request issuing, to improve resource utilization, throughput and fairness.

4.2.3 Methodology

We use GPGPUSim V3.2.2 [2], a cycle-accurate GPU microarchitecture simulator, to evaluate different CKE schemes. ***Table 5*** shows the baseline GPU architecture configuration, which is NVIDIA Maxwell-like. We have extensively modified GPGPUSim to issue warp instructions from kernels of multiple applications and share the same execution backend among them.

Table 6. Benchmarks

Benchmark	RF_oc	Smem_oc	Thread_oc	Cinst/ Minst	Req/ Minst	l1d_miss _rate	l1d_rsfail _rate	l1d_ MPKI	Type
hs (hotspot) [7]	98.4%	21.9%	58.3%	7	3	0.97	1.53	0.1	Com
pf (pathfinder) [7]	75.0%	25.0%	100.0%	6	2	0.99	0.00	0.1	Com
bp (backprop) [7]	56.2%	13.3%	100.0%	6	2	0.80	0.33	0.1	Com
sv (Spmv) [37]	75.0%	0.0%	100.0%	3	3	0.78	5.23	2.5	Mem
3m (3MM) [10]	56.2%	0.0%	100.0%	2	1	0.63	5.45	4.9	Mem
ks (kmeans) [7]	56.2%	0.0%	100.0%	3	17	1.00	7.96	99.9	Mem
ax (ATAX) [10]	56.2%	0.0%	100.0%	2	11	0.96	82.24	82.3	Mem

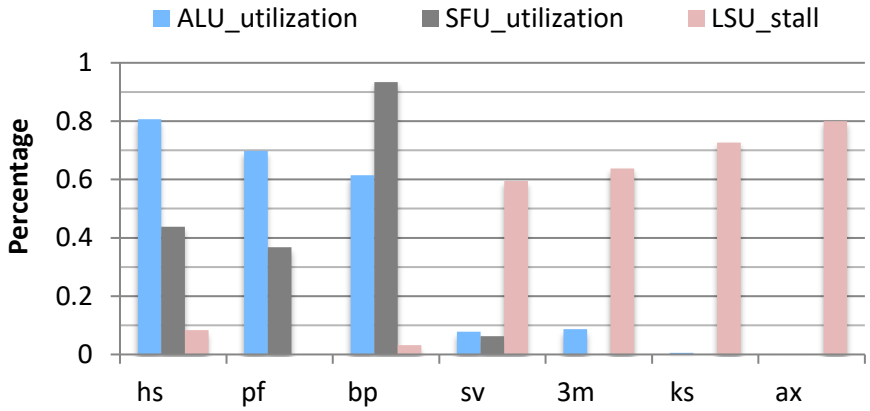


Figure 28. Computing resource utilization and LSU stalls.

We have studied GPU applications in image processing, math and scientific computing from Rodinia [8], Parboil [56] and Polybench [15]. Two-programmed workloads are constructed by paring different applications. Each workload runs for two million cycles and a kernel will restart if it completes before two million cycles, the same as in previous works [2][65].

In this study, we report the evaluation of system throughput (STP) and average normalized turnaround time (ANTT) which incorporates fairness [12]. STP is the sum of the normalized IPCs of co-run kernels (IPC of concurrent execution to that of isolated execution) and ANNT is the arithmetic average of normalized turnaround time.

4.2.4 Workload Characterization

In this section, we classify applications into compute-intensive and memory-intensive categories. We present motivational data regarding how utilization of computing units, the percentage of LSU (Load/Store Unit) stall cycles and the underlying memory/cache access behaviours vary across applications. Then we show that even the state-of-the-art SM sharing scheme, Warped-Slicer fails to address the negative interference among concurrent kernels and cannot achieve its predicted performance.

Table 6 and ***Figure 28*** together shows the characteristics of different benchmarks. First, ***Table 6*** shows the occupancy of static resources (e.g. registers, shared memory and the number of threads, which are allocated at the TB level). The static resource utilization can be improved if benchmarks with complementary requirements are running concurrently. For instance, while hs shows a high occupancy of registers and a relatively low occupancy of threads, ks shows the opposite trend, and therefore the utilization of registers and threads can be improved if hs and ks are running together. Second, as shown in ***Figure 28***, where benchmarks are arranged in decreasing order of ALU utilization, an inverse relationship exists between utilization of computing units and percentage of LSU stalls. Based on the percentage of LSU stalls, we classify benchmarks with less than 10% LSU stalls as *compute-intensive* (**Com**), and others as *memory-intensive* (**Mem**), indicated in the column 'Type' of ***Table 6***.

Table 6 also shows that compute-intensive kernels and memory-intensive ones have different memory/cache access behaviours. First, compute-intensive kernels show a high ratio regarding the number of compute instructions over that of memory ones, as indicated in the column 'Cinst/Minst', whereas memory-intensive ones have a lower 'Cinst/Minst'. For example, while 'Cinst/Minst' is 7 for hs, it is only 2 for 3m. Besides, the column 'Req/Minst' denotes the average number of requests per memory instruction. A low value of 'Req/Minst' indicates a high degree of memory request coalescing within a warp on average. As shown, memory-intensive kernels may have different degrees of memory request coalescing. For instance, 'Req/Minst' is 3 for sv and 17 for ks. And the other three columns 'l1d_miss_rate', 'l1d_rsfail_rate' and 'l1d_MPKI' denote miss rate, reservation failure per access and misses

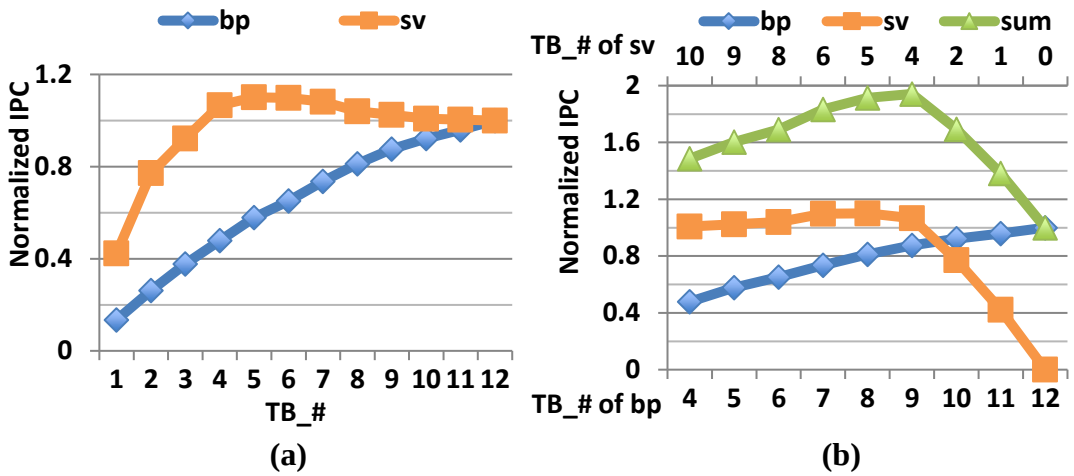


Figure 29. (a) Performance vs. increasing TB occupancy in one SM, (b) identify the performance sweet spot.

per kilo-instructions at L1 D-cache, respectively. Although compute-intensive kernels may show a high 'l1d_miss_rate', they have low 'l1d_MPKI', due to their streaming access in L1 D-cache and usage of shared memory. A high 'l1d_MPKI' indicates a large number of misses/accesses at L1 D-cache per kilo-instructions, such as for the cases ks and atx, and a high 'l1d_rsfail_rate' implies severe cache-miss-related resource congestion and thus memory pipeline stalls.

We create three classes of 2-kernel workloads by paring the benchmarks in ***Table 6***, namely **C+C**, **C+M** and **M+M**, where **C** stands for compute-intensive and **M** for memory-intensive.

4.2.5 Motivational Analysis

In this part, we show that the state-of-the-art SM-sharing scheme, Warped-Slicer, fails to address interference among co-run kernels and cannot achieve its predicted performance.

As shown in ***Figure 28***, when the percentage of LSU stalls increases, the computing resource utilization decreases, from compute-intensive applications to memory-intensive ones. Potentially, running compute-intensive applications together with memory-intensive ones on the same SM can improve the utilization of computing units. However, somewhat unexpected,

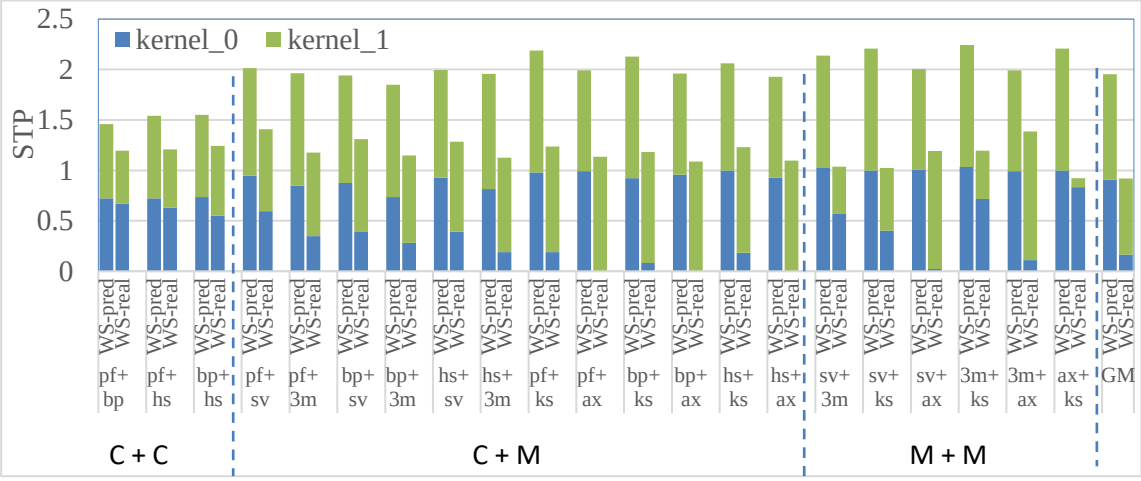


Figure 30. Performance gap between the prediction of Warped-Slicer and experimental results.

LSU stalls incurred by the memory-intensive applications will impose high penalty on the co-running compute-intensive ones, jeopardizing their performance.

Figure 29 illustrates the workflow of Warped-Slicer with the two-programmed workload bp+sv, where bp is compute-intensive and sv is memory-intensive. ***Figure 29***(a) shows the performance scalability curves of both benchmarks when they run in isolation. While the performance of bp near-linearly increases with a higher TB occupancy, the performance of sv first increases and then decreases with more TBs launched to an SM. Then as shown in ***Figure 29***(b), the scalability curves of bp and sv are used to identify the sweet point, i.e., the TB combination, where the performance loss of co-run kernels is minimized. For the case of bp+sv, the sweet point is (9, 4), i.e. 9 TBs from bp and 4 TBs from sv, and the predicted STP is 1.94.

However, ***Figure 30*** shows that the real STP of Warped-Slicer (WS-real) is much lower than its prediction (WS-pred). For bp+sv, the predicted STP is 1.94 while the actual STP is 1.31. The same trend can be observed in other workloads. Nevertheless, different combinations show different features regarding how individual kernels perform in co-run. First, for C+C workloads, STP is close to the prediction and co-run kernels show similar normalized IPC. Second, for C+M workloads, the memory-intensive kernel may dominate the usage of memory

pipeline and execution while the compute-intensive one shows significant performance loss because its memory requests suffer delays and cannot be timely served, so as its computation operations. Third, for M+M workloads, one kernel may suffer more than the other, due to their different memory/cache access behaviours. Therefore, it is crucial to reduce the interference among kernels in intra-SM sharing, especially in the memory pipeline and the memory subsystem, so as to serve memory requests from concurrent kernels in a fair manner, improve computing resource utilization and achieve better performance.

4.3 Overcome the Hurdle of Memory Pipeline Stalls

In Section 4.2.5, we demonstrated that memory pipeline stalls incurred by one kernel may affect other kernels running on the same SM and degrade the overall performance. Especially, compute-intensive kernels suffer a lot when running with memory-intensive ones. To address this issue, we investigate three methods to better accommodate memory requests from co-run kernels, improve computing resource utilization and thus the overall performance. The three methods are: 1) cache partitioning; 2) balance memory request issuing to prevent the starvation of any kernel in data access; and 3) limiting the number of inflight memory instructions from individual kernels to reduce L1 D-cache thrashing and memory pipeline stalls.

4.3.1 Cache Partitioning

In this part, we show that simply applying cache partitioning cannot effectively improve the overall performance of intra-SM sharing on GPUs.

Figure 31 illustrates the effectiveness of cache partitioning, where 'WS' denotes TB partition with Warped-Slicer and 'WS-L1D Partition' is for WS plus L1 D-cache partitioning, in which UCP (Utility-based Cache Partitioning) [49] is adopted. First, ***Figure 31***(a) shows that on average, L1 D-cache partitioning fails to improve STP across all three workload classes. Two workloads from each class are selected for further investigation. Concretely, pf+bp and bp+hs are C+C workloads, bp+sv and bp+ks are C+M, and sv+ks and sv+ax are M+M. Among them, there is not much STP variation for pf+bp and bp+sv; and bp+hs, bp+ks and sv+ks show performance degradation with L1 D-cache partitioning; only sv+atx obtains performance improvement.

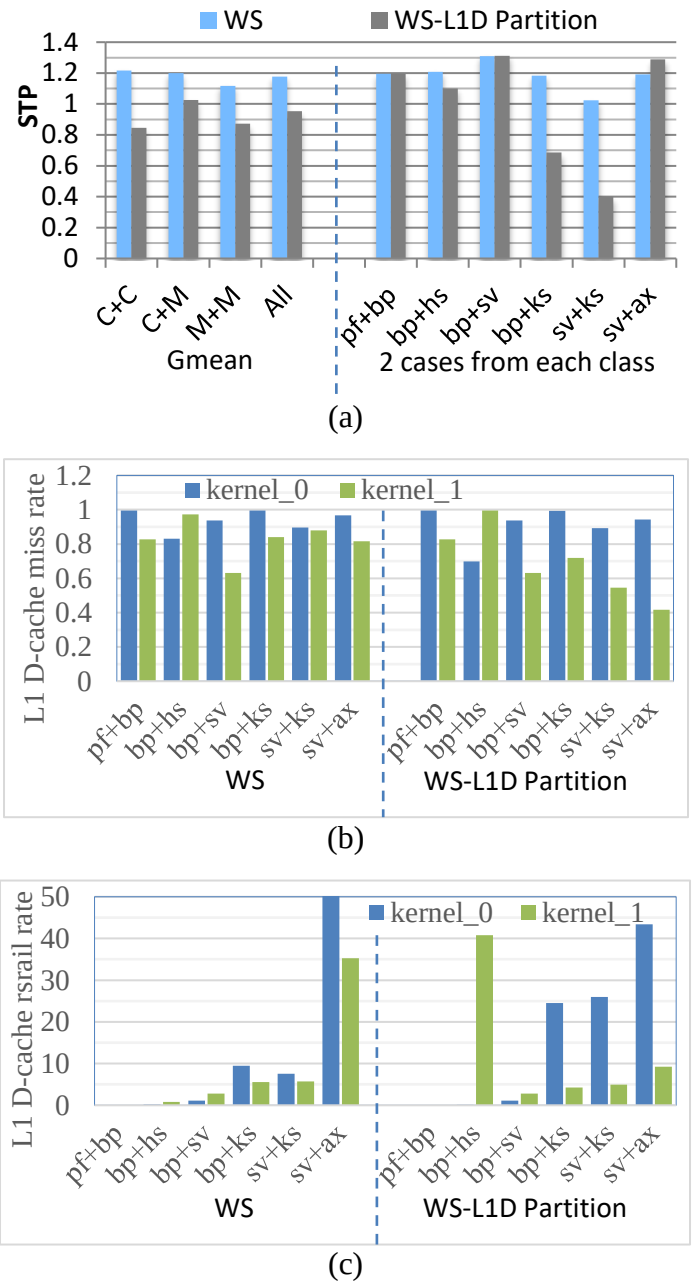


Figure 31. Effectiveness of Cache Partitioning: (a) STP; (b) L1 D-cache miss rate; and (c) L1 D-cache rsfail rate.

We check L1 D-cache efficiency and cache-miss-related resource congestion to better understand how performance is affected by cache partitioning. ***Figure 31*** (b) and (c) presents L1 D-cache miss rate and rsfail rate (reservation failures per access) of individual kernels in

the selected workloads. First, pf+bp and bp+sv do not have much variation in L1 D-cache miss rate and rsfail rate, corresponding to their similar performance with and without L1 D-cache partitioning. Second, although bp in bp+hs, ks in bp+ks and ks in sv+ks has a lower L1 D-cache miss rate in 'WS-L1D Partition' than WS, the other kernel, namely hs in bp+hs, bp in bp+ks and sv in sv+ks, suffers from a much higher L1 D-cache rsfail rate, i.e. more reservation failures per access, because a smaller portion of L1 D-cache is assigned to it according to UCP while a cache slot still needs to be allocated for an outstanding miss. The combined effect of reduced L1 D-cache miss rate of one kernel and a higher rsfail rate of the other leads to the performance degradation. Third, for sv+ax, ax obtains lower L1 D-cache miss rate and rsfail rate, and the reduction in reservation failures of ax even benefits the co-running kernel, sv, resulting in improved performance of sv+ax with 'WS-L1D Partition'.

As demonstrated, although cache partitioning may help reduce the miss rate of a kernel in intra-SM sharing, it does not necessarily reduce memory pipeline stalls. So it remains important to examine other approaches to mitigate memory pipeline stalls and improve the overall performance.

4.3.2 BMI: Balanced Memory Request Issuing

In this section, we present the idea of balanced memory request issuing in intra-SM sharing.

As shown in ***Table 6***, compute-intensive kernels have a higher ratio regarding the number of compute instructions over memory instructions, indicated by 'Cinst/Minst', than memory-intensive ones. Also, compute-intensive kernels have lower requests per memory instruction, indicated by 'Req/Minst'. When multiple kernels concurrently run on the same SM, they compete for the same memory pipeline. As a memory-intensive kernel has more memory instructions in nature, it has a higher probability to access LSU (Load/Store Unit) if there is no dedicated memory instruction issuing management. When the compute-intensive kernel needs to issue a memory instruction, however, it has to wait until the LSU becomes available. Due to the high 'Req/Minst' of memory-intensive kernels, the waiting time can be quite long especially when the memory-intensive kernel has a low L1 D-cache hit rate.

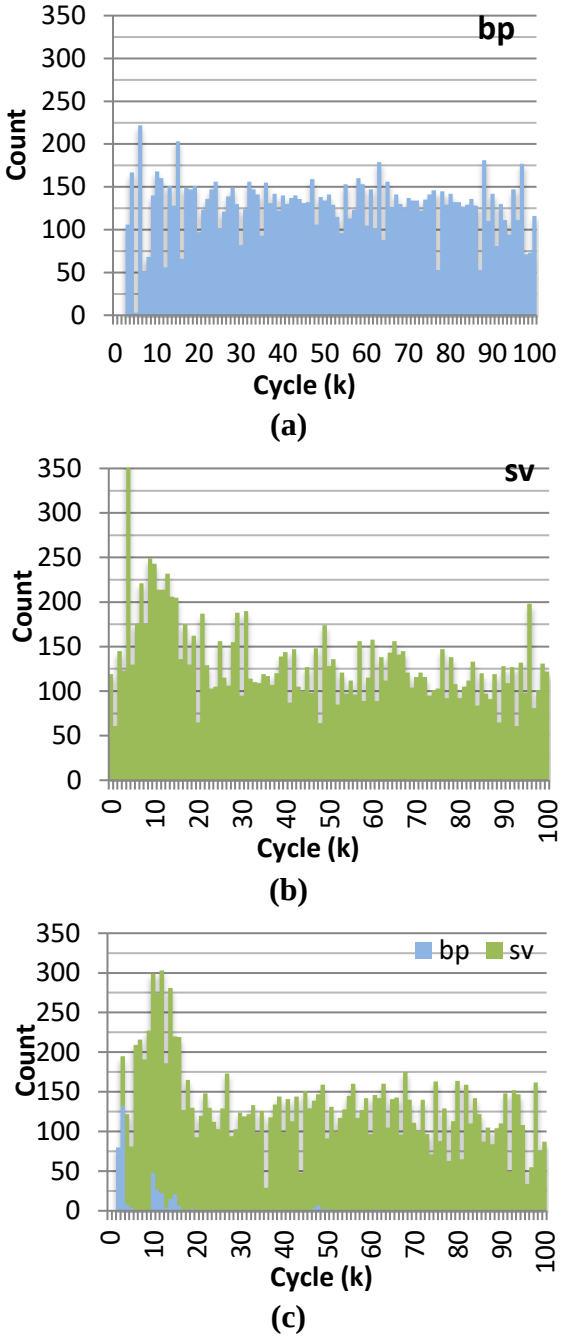


Figure 32. L1 D-cache accesses: (a) bp executes in isolation; (b) sv executes in isolation; (c) bp and sv run concurrently.

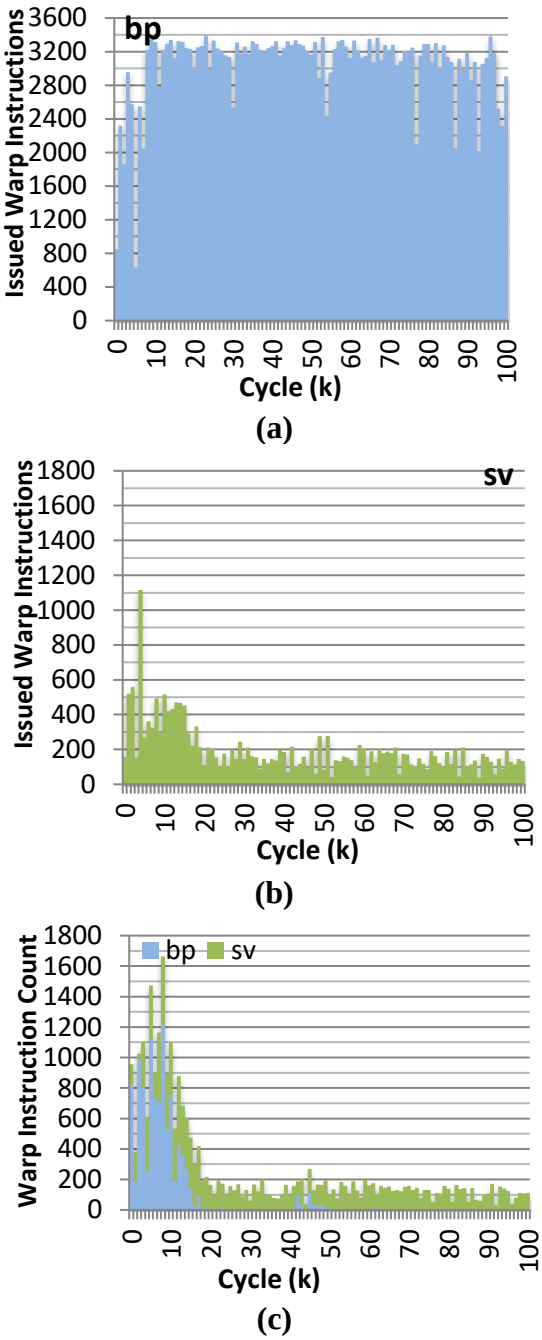


Figure 33. Warp Instruction Issuing: (a) bp executes in isolation; (b) sv executes in isolation; (c) bp and sv run concurrently.

Moreover, the compute-intensive kernel may lose the competition again due to the warp scheduling policy even when the LSU is available.

Figure 32 shows the number of L1 D-cache accesses with a sampling interval of 1K cycles for a total of 100K cycles for the workload bp+sv, where bp is compute-intensive and sv is memory-intensive. ***Figure 32*** (a) and (b) show that both bp and sv have a considerable amount of accesses every 1K cycles when either runs in isolation. However, as shown in ***Figure 32*** (c), when bp runs concurrently with sv, sv dominates L1 D-cache accesses and bp starves. As a result, bp cannot get its memory requests accommodated timely and in turn not able to execute data-dependent computation instructions, leading to the degradation of the overall performance. Also, as shown in ***Figure 33***, although bp has a large number of instructions per 1K cycles when running in isolation, it significantly suffers when running with sv. Correspondingly, ***Figure 33***(c) shows the number of issued warp instructions with a sampling interval of 1K cycles for a total of 100K cycles for the workload bp+sv. ***Figure 33*** (a) and (b) show that both bp and sv have a considerable amount of warp instructions every 1K cycles when either runs in isolation. As discussed in Section 4.2.4, bp has a high ratio of computation instructions while sv has a relatively high of memory instructions. When bp runs concurrently with sv, since sv dominates L1 D-cache accesses, bp cannot retrieve data to execute computation instructions in time, it also experiences starvation in warp instruction issuing, as shown in ***Figure 32*** (c).

To resolve the problem that one kernel starves from failing to access the memory subsystem, we propose balanced memory request issuing (BMI). One variant of BMI is called RBMI, where memory instructions from concurrent kernels are issued in a loose round-robin manner. However, since one warp memory instruction may result in multiple memory requests and different kernels show different 'Req/Minst' (***Table 6***), RBMI cannot ensure balanced memory access among concurrent kernels. To overcome this problem, we propose quota-based memory instruction issuing, named QBMI, where the memory instruction quotas of concurrent kernels are calculated with the formula below:

$$Quota_i = \frac{LCM(\frac{Req_0}{Minst_0}, \frac{Req_1}{Minst_1}, \dots, \frac{Req_n}{Minst_n})}{\frac{Req_i}{Minst_i}}$$

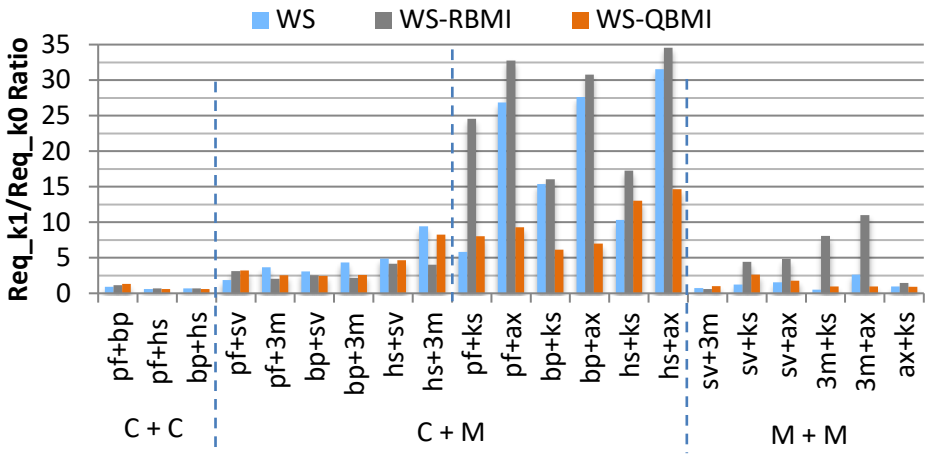


Figure 34. The ratio of the total number of memory requests from kernel_1 over that from kernel_0.

where $Quota_i$ is the quota for kernel i, LCM denotes Least Common Multiple and $\frac{Req_j}{Minst_j}$ is the average number of requests per memory instruction for kernel j. Therefore, the higher 'Req/Minst' of a kernel, the lower quota is assigned to it. The priority of a kernel to issue a memory instruction is based on its current quota and the more its quota, the higher its priority. Each time a memory instruction is issued from a kernel, its quota is decremented by 1. When the quota of any kernel reaches zero, a new set of quotas, calculated with the most recent values of 'Req/Minst', will be added to the current quota value of concurrent kernels, so as to eliminate the scenario where a kernel with zero quota cannot issue memory instructions even when there is no ready memory instruction from any other co-run kernel. Specifically, 'Req/Minst' of a kernel is updated every 1024 memory requests issued by it. The sampling interval of 1024 accesses works well as the metric 'Req/Minst' is relatively stable throughout the execution of a GPU kernel.

Figure 34 shows the ratio regarding the total number of memory requests from kernel 1 over that from kernel 0 for 2-kernel concurrent execution, where WS-RBMI denotes TB partition with Warped-Slicer plus RBMI and WS-QBMI is Warped-Slicer plus QBMI. First, there is not much variation for C+C workloads, since the two compute-intensive kernels both have high 'Cinst/Minst' and low 'Req/Minst'. Second, for C+M workloads, RBMI and QBMI

both achieve more balanced memory access when the memory-intensive kernel has a relatively low 'Req/Minst', similar to that of the co-run compute-intensive kernel, and six workloads from pf+sv to hs+3m fall into this category. However, when the memory-intensive kernel has a high 'Req/Minst', RBMI fails to balance memory accesses, as for the six C+M workloads from pf+ks to hs+ax, where ks and ax have a much higher number of requests per memory instruction than the co-run compute-intensive kernel (***Table 6***). Likewise, for M+M workloads, there is not much variation when the two kernels have similar 'Req/Minst' and RBMI does not perform well when one kernel has a much higher value of 'Req/Minst' than the other while QBMI remains effective. Since QBMI can achieve balanced memory access more effectively than RBMI, we adopt QBMI in the rest of the paper.

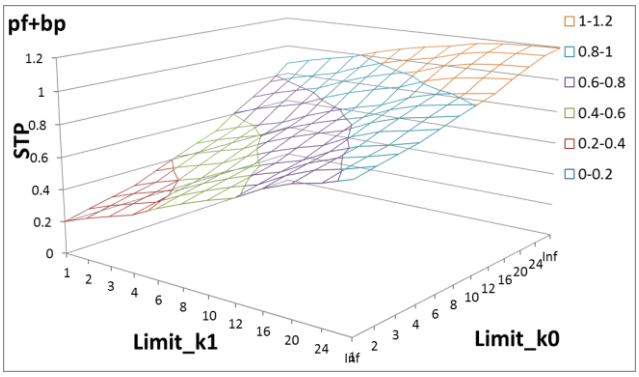
4.3.3 MIL: Memory Instruction Limiting

In this section, we highlight that the overall performance can be improved by explicitly limiting the number of in-flight memory instructions. Besides, different kernels are in favour of different memory instruction limiting numbers.

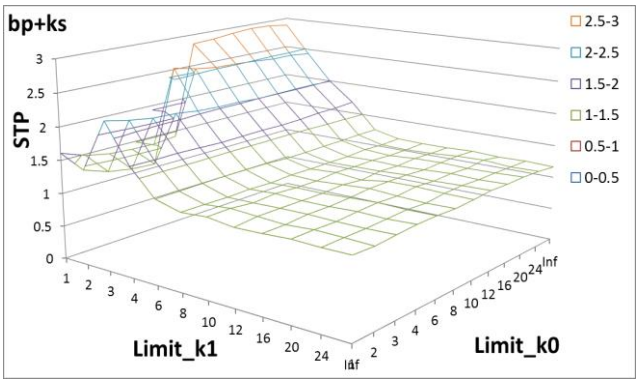
Although BMI can help achieve more balanced memory accesses and get a kernel's requests served more timely, it does not necessarily reduce memory pipeline stalls incurred by a memory-intensive kernel and the co-run kernels may still suffer from such penalties. In the meanwhile, limiting the number of in-flight memory instructions of a kernel is an effective way to reduce memory pipeline stalls and improve L1 D-cache efficiency. To exploit this factor, we propose MIL (Memory Instruction Limiting) for intra-SM sharing and we investigate two variants of MIL: SMIL (static MIL) and DMIL (dynamic MIL).

4.3.3.1 SMIL: Static Memory Instruction Limiting

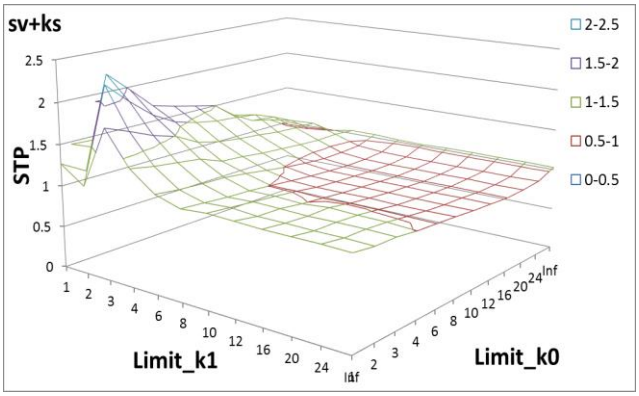
In this section, we illustrate the effectiveness of static memory instruction limiting (SMIL). For SMIL, we ran simulations for all combinations regarding the number of in-flight memory instructions that can be issued from individual kernels in intra-SM sharing. Specifically, we vary the in-flight memory instruction limiting number on kernel 0 from 1 to 24, and symmetrically for kernel 1. The simulation point of no such a limitation (Inf) for each kernel is also examined.



(a) pf+bp



(b) bp+ks



(c) sv+ks

Figure 35. STP with varied memory instruction limiting numbers on kernel 0 and kernel 1 for workloads from different classes: (a) C+C workload: pf+bp; (b) C+M workload: bp+ks; (c) M+M workload: sv+ks.

We use one representative workload from each class to demonstrate how SMIL performs. ***Figure 35*** shows the performance with varied memory instruction limiting numbers

on kernel 0 and kernel 1. In the figures, the right horizontal axis (Limit_k0) denotes the memory instruction limiting number on kernel 0 while the left one (Limit_k1) is for kernel 1, and the vertical axis indicates STP.

Figure 35 (a) shows that for the C+C workload, pf+bp, with a fixed Limit_k1, the performance increases with a larger Limit_k0 and it is similar when varying Limit_k1 with Limit_k0 fixed. Therefore, there is no need to limit the number of inflight memory instructions for co-run compute-intensive kernels. ***Figure 35*** (b) shows the case of a C+M workload, bp+ks, in which the overall performance suffers from a large number of in-flight memory instructions issued by kernel 1, ks. When Limit_k1 is at least 8, the performance is low with varied Limit_k0 and Limit_k1. When Limit_k1 is small (smaller than 8), a better performance can be achieved with a large Limit_k0, due to the improved L1 D-cache locality resulted from the combined effect of in-flight memory instruction limiting and underlying warp scheduling policy GTO. ***Figure 35***(c) presents the case of sv+ks, an M+M workload, in which the performance remains low when Limit_k1 is at least 8. However, different from bp+ks, the overall performance first increases and then decreases with a larger Limit_k0 when Limit_k1 is small (smaller than 8). As a result, an optimal point exists in terms of the peak performance and it is (3, 1) for sv+ks, indicating the highest performance occurs when Limit_k0 is 3 and Limit_k1 is 1.

As illustrated, limiting the number of in-flight memory instructions from the memory-intensive kernel effectively improves the overall performance. The compute-intensive kernel can have a better chance to access memory subsystem and get requests served timely when memory pipeline stalls are reduced with fewer in-flight memory instructions from the co-run memory-intensive kernel. In the meanwhile, the performance of the memory-intensive kernel also increases due to the improved L1 D-cache efficiency. However, since different kernels have different memory access features, indicated by metrics like ‘Req/Minst’ and ‘l1d miss rate’ (***Table 6***), different workloads have different optimal limiting numbers for concurrently running kernels.

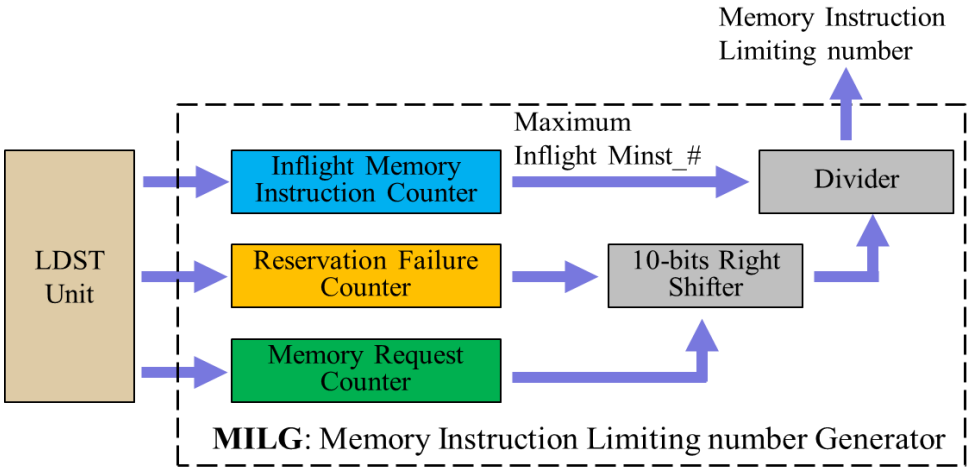


Figure 36. Organization of a Memory Instruction Limiting number Generator (MILG).

4.3.3.2 DMIL: Dynamic Memory Instruction Limiting

Although SMIL can effectively improve the overall performance, it requires re-profiling whenever there are updates in architecture, application optimization and input size. Besides, it cannot cope well with application behaviour changing from phase to phase. To overcome these problems, we propose dynamic MIL (DMIL) to adapt the in-flight memory instruction limiting numbers at run-time.

As discussed in Section 4.2.1, cache-miss-related resources, including a cache line slot, a MSHR and miss queue entry, are allocated for an outstanding miss. If any of the required resources is unavailable, there will be *reservation failures*, resulting in memory pipeline stalls. And memory pipeline stalls incurred by one kernel will affect other concurrent kernels, further reducing computing resource utilization and degrading performance. Therefore, we use the number of *reservation failures per memory request* as the indicator to check how severe cache contention and cache-miss-related resource congestion is. The goal of limiting the number of in-flight memory instructions of a kernel is to achieve as few reservation failures per memory request as possible.

Figure 36 shows the organization of a memory instruction limiting number generator (MILG), which has one in-flight memory instruction counter, one reservation failure counter,

one memory request counter and one 10-bit right shifter. In our experiments, a new memory instruction limiting number for a kernel is produced for every 1024 memory requests issued by this kernel. The sampling interval of 1024 accesses works well in capturing the phase behaviours. In a MILG, the 10-bit right shifter is used to calculate reservation failures per memory request. From the in-flight memory instruction counter, we can get the maximum number of inflight memory instruction in the last sampling interval. And the memory instruction limiting number is generated using the formula below:

$$MIL\_number = Max\_Inflight\_Minst\_\# \,/\, (Rev\_Fail\_\#>>10)$$

The number of inflight memory instructions from a kernel is reduced when there are more than one reservation failures per memory request and the ultimate goal is to achieve at most one reservation failure per memory request. To avoid the scenario that a kernel is prohibited to issue memory instructions, the policy that at least one inflight memory instruction from a kernel is incorporated.

Since each kernel has its own MILG, for our created 2-kernel workloads, there are two MILGs on each SM. And it is flexible to extend for more kernels. Since there are MILGs in each SM, we refer to this design as local DMIL. Although it is possible to reduce the hardware cost by deploying global DMIL, which monitors concurrent kernel execution on one SM and broadcasts the generated results to others, global DMIL requires all SMs run the same pair of kernels. Due to the inflexibility of global DMIL, we stick to local DMIL in this study.

4.3.4 QBMI vs. DMIL

As discussed in Section 4.3.2 and 4.3.3, QBMI can balance memory accesses of concurrent kernels and DMIL can boost performance by reducing memory pipeline stalls incurred by memory-intensive kernels. In this part, we compare the performance impact of QBMI and DMIL, and investigate the integration of the two.

Figure 37 illustrates how QBMI and DMIL perform when Warped-Slicer is used for TB partition and QBMI+DMIL denotes the combination of the two. ***Figure 37***(a) shows the system throughput. First, WS-QBMI and WS-DMIL achieve similar performance for C+C

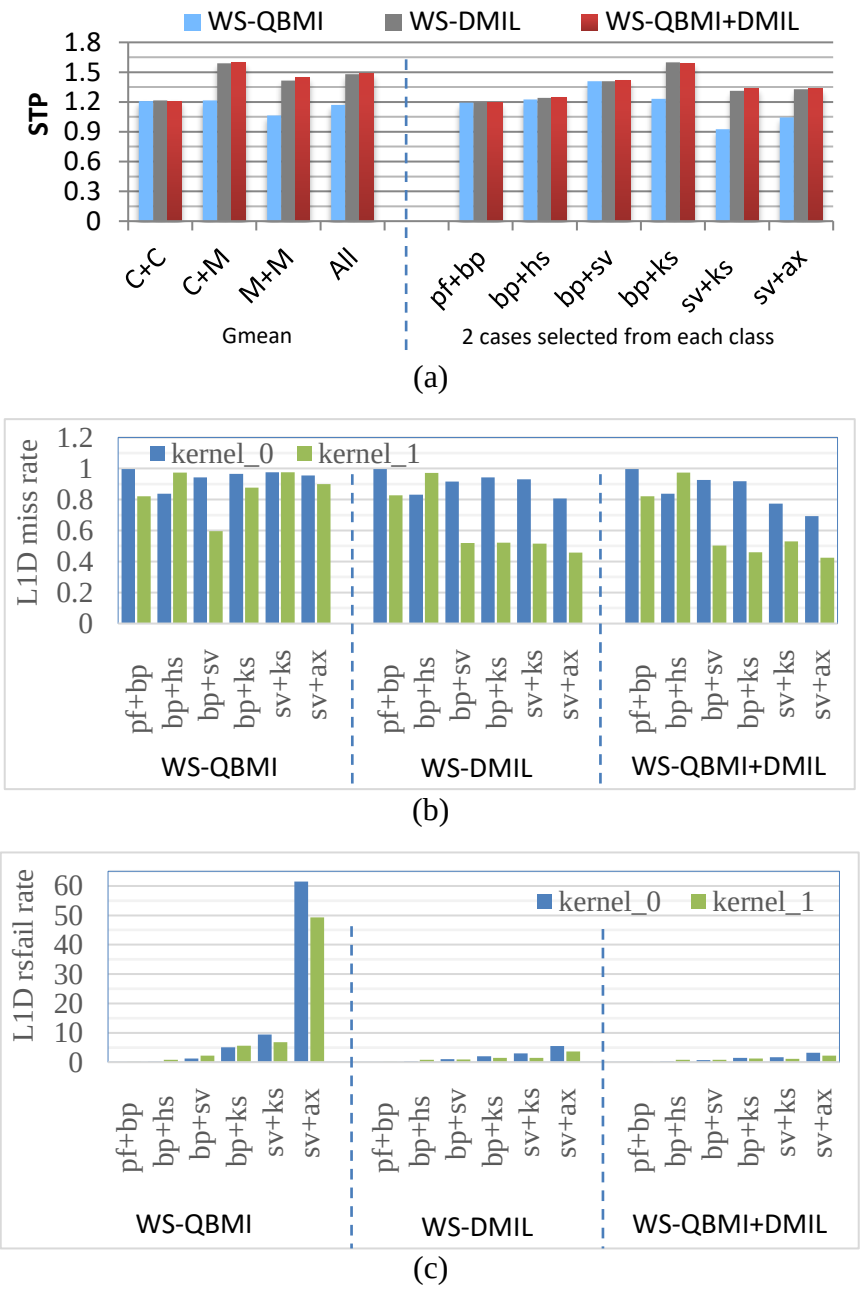


Figure 37. Performance impact of QBMI and DMIL.

workloads since compute-intensive kernels have high 'Cinst/Minst' and low memory pipeline stalls. Second, on average, WS-DMIL outperforms WS-QBMI for C+M and M+M workloads with improved L1 D-cache efficiency and further reduced memory pipeline stalls. For instance, compared to WS-QBMI, WS-DMIL effectively reduces L1 D-cache miss rate of ks from 0.88

to 0.52 in the C+M workload bp+ks and from 0.98 to 0.52 in the M+M workload sv+ks (***Figure 37***(b)). ***Figure 37***(c) shows that WS-DMIL has a lower L1 D-cache rsfail rate than WS-QBMI, indicating fewer memory pipeline stalls.

While it is tempting to integrate QBMI and DMIL to reap the benefits of both, ***Figure 37***(b) and (c) show that the improvement from WS-QBMI+DMIL over WS-DMIL is minor regarding L1 D-cache efficiency and memory pipeline stalls, resulting its slightly better performance than WS-DMIL, shown in ***Figure 37***(a). Therefore, we report how QBMI and DMIL perform separately in the evaluation.

4.4 Experimental Results and Analysis

In this section, we conduct experimental analysis on our schemes and investigate how they improve the performance of the two state-of-art intra-SM sharing techniques, Warped-Slicer [71] and SMK [65], both targeting TB (Thread-Block) partition, as described in Section 4.1.

WS: Warped-Slicer, which enforces TB partition using scalability curves generated by profiling individual kernels in isolation [71].

WS-QBMI: Our proposed quota-based balance memory request issuing (QBMI) is applied to WS.

WS-DMIL: Our proposed dynamic memory instruction limiting (DMIL) is applied to WS.

SMK-(P+W): SMK-(P+W) in [65], which enforces TB partition based on fairness of static resources allocation and periodically allocates warp instruction quotas for concurrent kernels with profiling each one in isolation. In SMK-(P+W), a kernel will stop issuing instructions if it runs out of quota and a new set of quotas will be assigned only when quotas of all kernels equal zero. Since the warp instruction quota allocation in SMK-(P+W) and our proposed QBMI/DMIL are mutually exclusive, we apply our schemes to SMK-P and compare them with SMK-(P+W).

SMK-(P+QBMI): QBMI is applied to SMK-P.

SMK-(P+DMIL): DMIL is applied to SMK-P.

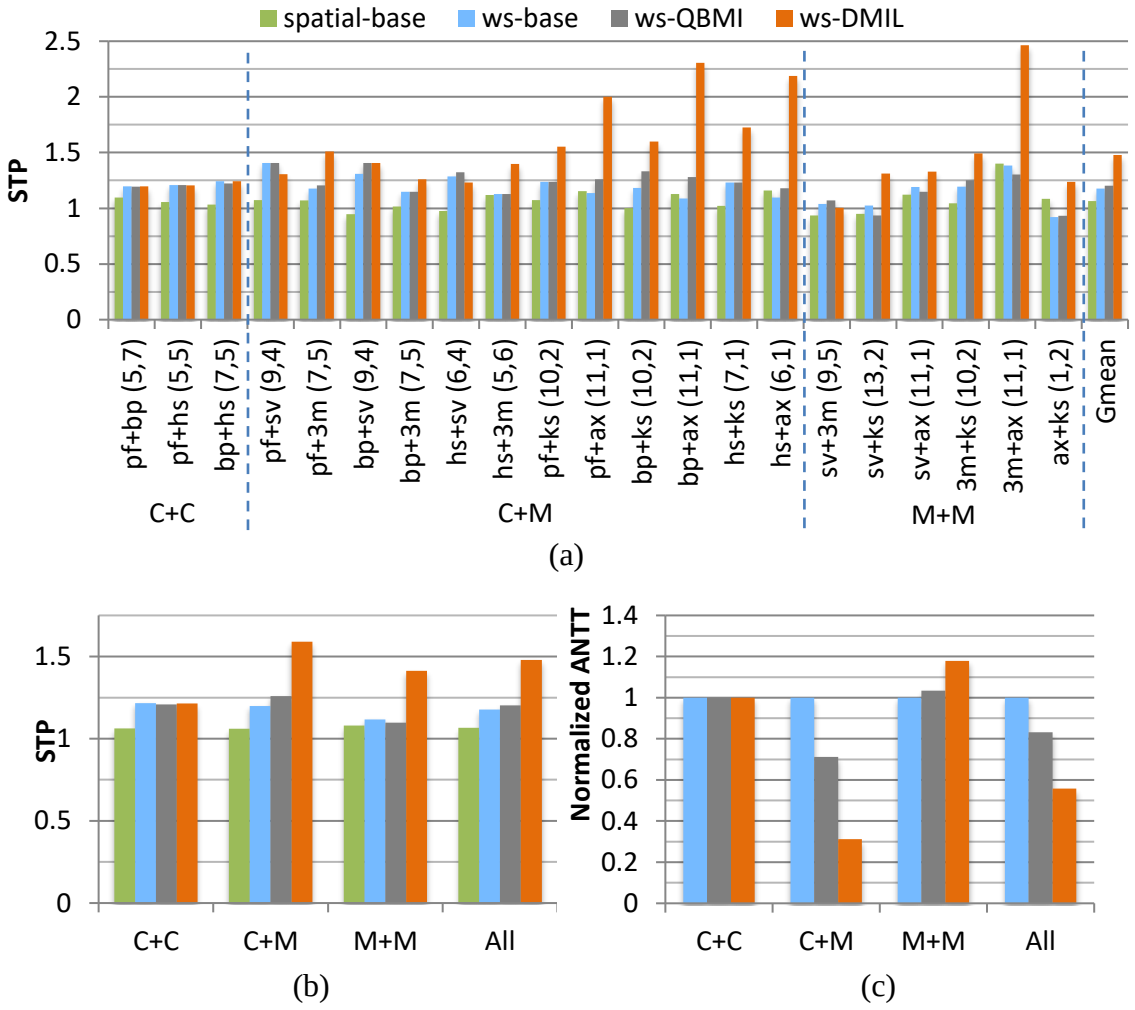


Figure 38. Effectiveness of QBMI and DMIL on top of Warped-Slicer: (a) STP of 2-kernel workloads; (b) STP of different workload classes; (c) Normalized ANTT.

4.4.1 Performance Evaluation and Analysis

4.4.1.1 Comparison with Warped-Slicer

In this part, we illustrate how our proposed QBMI and DMIL perform when Warped-Slicer is used for TB partition.

a) System Throughput and Fairness

Figure 38(a) and (b) show the system throughput (STP) of WS, WS-QBMI and WS-DMIL, and spatial multitasking (*Spatial*) is shown as a reference. The TB partition of a workload is

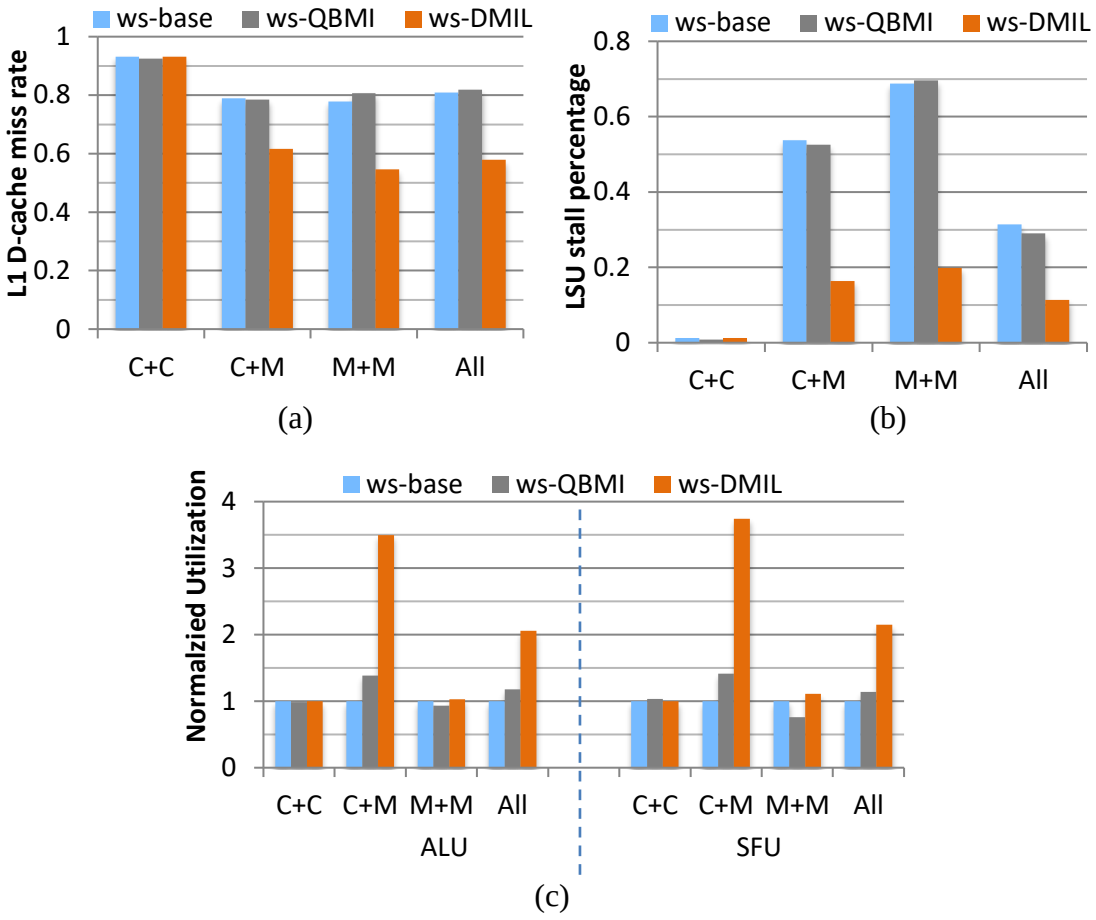


Figure 39. Effectiveness of QBMI and DMIL on top of Warped-Slicer: (a) L1 D-cache miss rate; (b) percentage of LSU stall cycles; (c) computing resource utilization.

also denoted in ***Figure 38***(a). For instance, 'pf+bp (5,7)' indicates that there are 5 TBs from pf and 7 TBs from bp. We have the following observations. First, WS performs better than Spatial on average with a better resource utilization within an SM, consistent with prior works [65][71]. However, there are cases for which the interference among kernels is high and Spatial outperforms WS, like hs+ax. Second, WS, WS-QBMI and WS-DMIL have similar STP for C+C workloads where there are almost no memory pipeline stalls. Third, while WS-QBMI and WS-DMIL outperform WS for C+M workloads, WS-DMIL has a much higher STP due to further reduced memory pipeline stalls and improved L1 D-cache efficiency. Moreover, WS-DMIL remains effective for M+M workloads while WS-QBMI fails to improve STP. On

average, STP is 1.06 from Spatial, 1.17 from WS, 1.20 from WS-QBMI and 1.48 from WS-DMIL. Thus WS-QBMI and WS-DMIL improve the performance of WS by 2.6% and 26.5%, respectively.

In addition to system throughput, we use ANTT (the lower the better) to measure the fairness [12], shown in ***Figure 38***(c). First, there is not much variation of ANTT for C+C workloads. Second, WS-QBMI and WS-DMIL greatly improve fairness over WS for C+M workloads. However, WS-QBMI and especially WS-DMIL show worse ANTT than WS for M+M workloads because one kernel may obtain much higher performance improvement than the other with its improved L1 D-cache efficiency and thus fairness is sacrificed in exchange for system throughput. Nevertheless, WS-QBMI and WS-DBMIL outperforms WS by 16.9% and 44.2% in terms of ANTT, on average.

b) L1 D-cache Miss Rate and LSU Stalls

In this section, we show that our proposed schemes achieve high L1 D-cache efficiency and significantly reduce memory pipeline stalls. ***Figure 39*** (a) and (b) show that without taking care of interference among concurrent kernels, WS suffers from a high L1 D-cache miss rate and a lot of LSU stalls in C+M and M+M workloads. WS-QBMI experiences similar L1 D-cache miss rate and LSU stalls to WS. In contrast, WS-DMIL consistently demonstrates lower L1 D-cache miss rates and fewer LSU stalls than both WS and WS-QBMI.

c) Computing Resource Utilization

Figure 39(c) presents the computing resource utilization, and together with ***Figure 39***(b), they show an inverse correlation exists between memory pipeline (LSU) stalls and computing resource utilization. For instance, when WS, WS-QBMI and WS-DMIL all have low percentage of LSU stalls, they achieve similar ALU/SFU utilization for C+C workloads. When WS-QBMI and WS-DMIL show fewer LSU stalls than WS for C+M workloads, they gain higher computing resource utilization. And WS-DMIL remains effective to improve compute resource utilization for M+M workloads, in which memory instructions dominate execution.

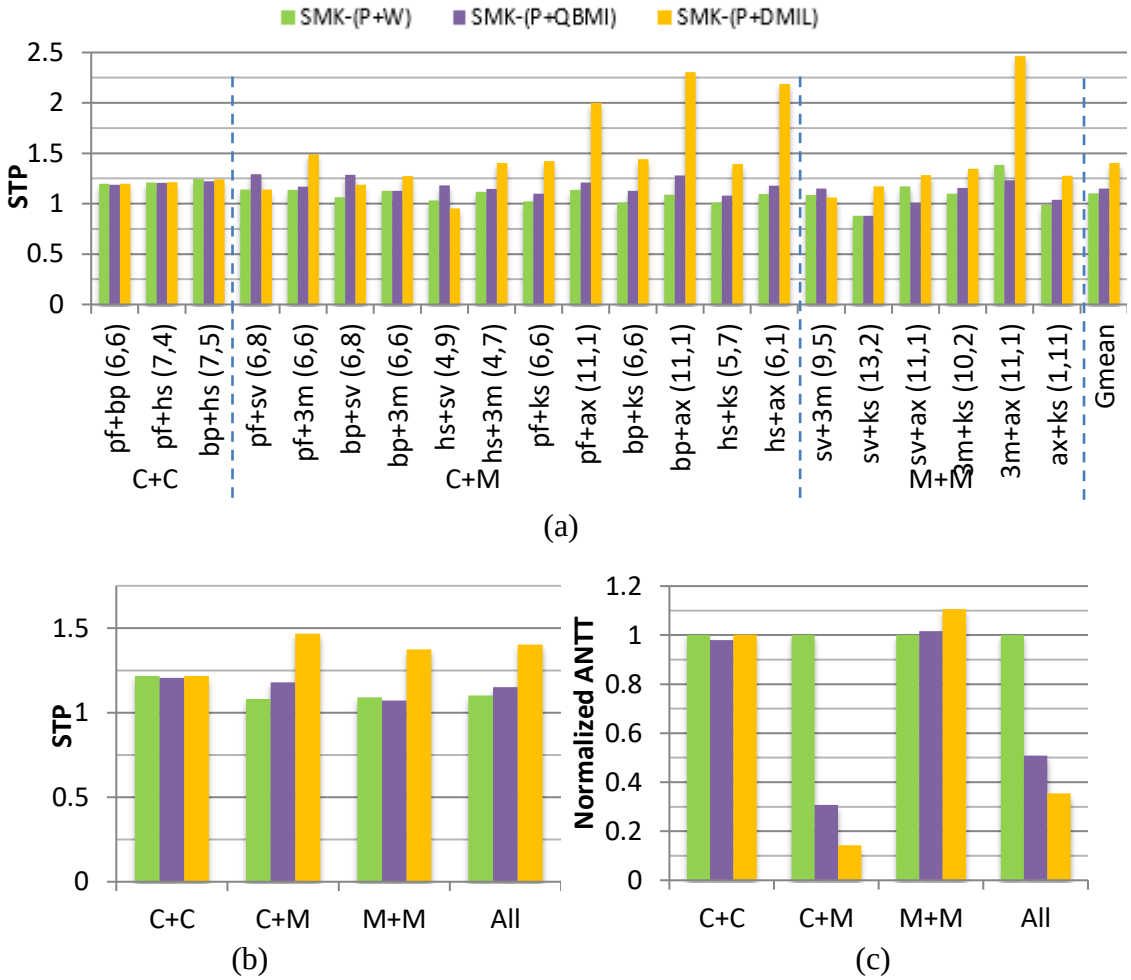
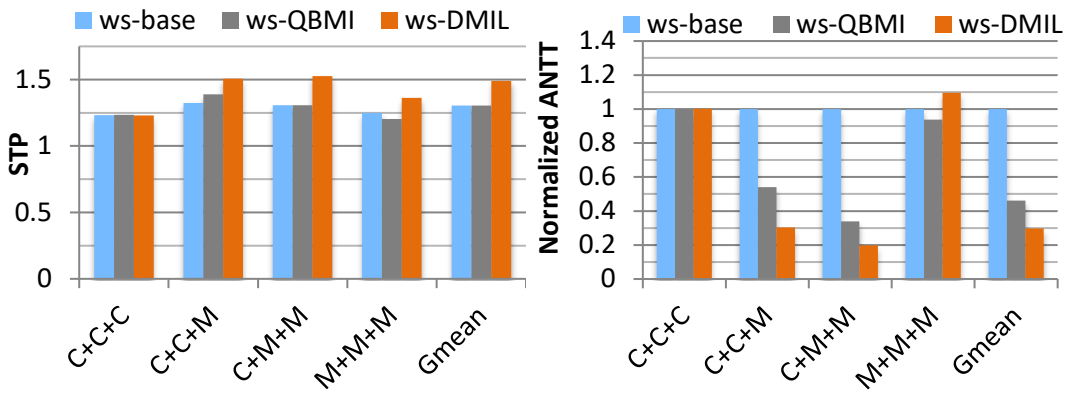


Figure 40. Effectiveness of QBMI and DMIL on top of SMK: (a) STP of 2-kernel workloads; (b) STP of different workload classes; (c) Normalized ANTT.

4.4.1.2 Comparison with SMK

Besides Warped-Slicer, we also investigate how QBMI and DMIL perform when SMK is used for TB partition. ***Figure 40*** shows STP of SMK-(P+W), SMK-(P+QBMI) and SMK-(P+DMIL). The TB partition of a workload generated by SMK is also denoted in ***Figure 40***(a). We have similar observations to those when Warped-Slicer is used. First, all three schemes have similar STP for C+C workloads. Second, SMK-(P+QBMI) and SMK-(P+DMIL) outperform SMK-(P+W) for C+M workloads, while SMK-(P+DMIL) has a much higher STP. Third, SMK-(P+DMIL) remains effective for M+M workloads. On average, STP is 1.10 from



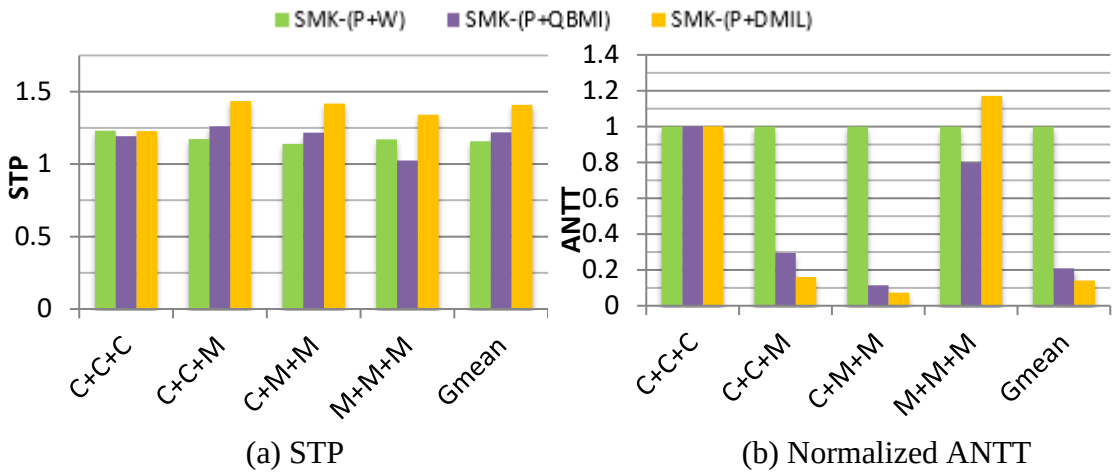
(a) STP (b) Normalized ANTT

Figure 41. Effectiveness of QBMI and DMIL in 3-kernel concurrent execution on top of Warped-Slicer.

SMK-(P+W), 1.15 from SMK-(P+QBMI) and 1.40 from SMK-(P+DMIL). Thus SMK-(P+QBMI) and SMK-(P+DMIL) boost the performance of SMK-(P+W) by 4.4% and 27.2%, respectively. As shown in ***Figure 40***(c) SMK-(P+QBMI) and SMK-(P+DMIL) outperforms SMK-(P+W) by 50.7% and 35.4% in terms of ANTT. The improvements of SMK-(P+QBMI) and SMK-(P+DMIL) are due to higher L1 D-cache efficiency, reduced LSU stalls and higher computing unit utilization.

4.4.2 More Kernels in Concurrent Execution

In this part, we demonstrate that our proposed schemes have good scalability and remain effective when more than two kernels concurrently on an SM. As described in Section 4.3.2 and 4.3.3, the proposed QBMI and DMIL are general and not restrained by the number of concurrent kernels. We evaluate all the combinations of 3-kernel workloads and have similar observations to those on 2-kernel workloads, as shown in ***Figure 41***. First, WS, WS-QBMI and WS-DMIL have similar STP and ANTT when all kernels are compute-intensive, indicated by ‘C+C+C’. Second, WS-QBMI and WS-DMIL outperform WS for C+C+M workloads. Third, WS-DMIL continues to improve STP for C+M+M and M+M+M workloads, where there are multiple memory-intensive kernels, but sacrifices fairness for system throughput for M+M+M workloads. On average, WS-QBMI and WS-DMIL improve STP of WS by 1.2%



(a) STP (b) Normalized ANTT

Figure 42. Effectiveness of QBMI and DMIL in 3-kernel concurrent execution on top of SMK.

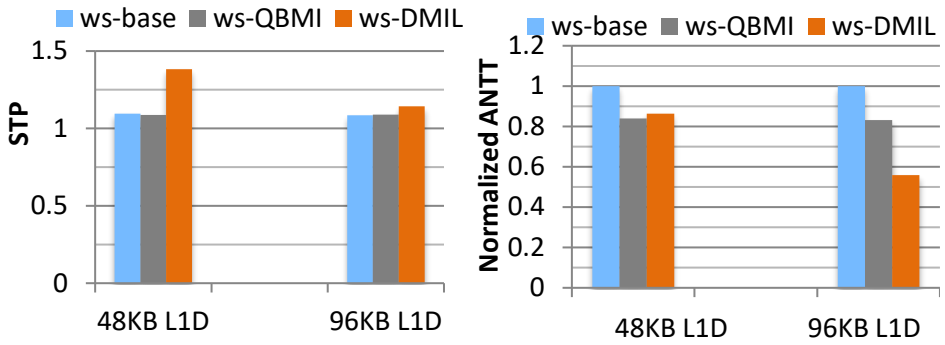
and 14.3%, respectively. In terms of ANTT, WS-QBMI and WS-DMIL outperform WS by 53.9% and 70.7%.

Besides Warped-Slicer, we also investigate when SMK is used for TB partition in 3-kernel concurrent execution. As shown in ***Figure 42***, SMK-(P+QBMI) and SMK-(P+DMIL) improve the average STP of SMK-(P+W) by 5.5% and 21.9%; for ANTT, SMK-(P+QBMI) and SMK-(P+DMIL) outperform SMK-(P+W) by 79.1% and 85.9%, respectively.

4.4.3 Sensitivity Study

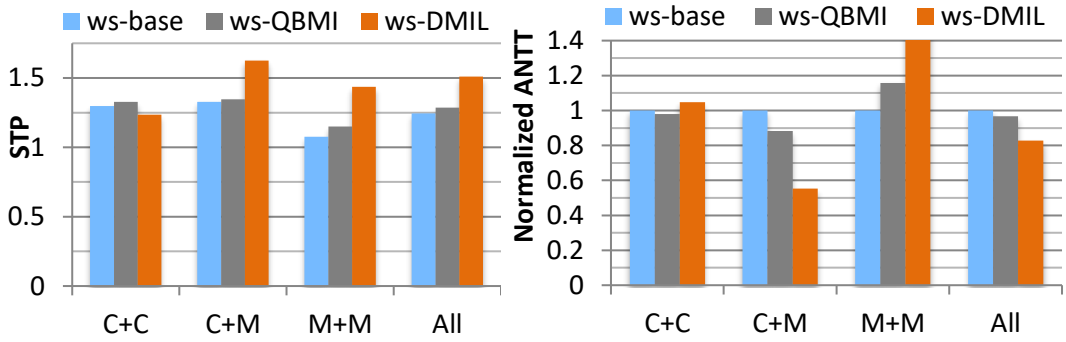
Sensitivity to L1 D-cache Capacity: ***Figure 44*** shows the performance under various L1 D-cache sizes. WS-DMIL constantly improves STP and ANTT. Although WS-QBMI may not achieve a better STP than WS, it improves fairness with a better ANTT. On average, WS-QBMI improves ANTT by 15.9% (16.9%) over WS; and WS-DMIL outperforms WS with 26.2% (5.3%) higher STP and 13.7% (44.2%) better ANTT, on a 48KB (96KB) L1 D-cache.

Sensitivity to Warp Scheduling Policy: besides the default GTO (Greedy-Then-Oldest) warp scheduling policy used in the prior experiments, we investigate how QBMI and DMIL perform when LRR (Loose Round Robin) is used for warp scheduling. As shown in ***Figure 43***, on average, WS-QBMI and WS-DMIL boost the average STP of WS by 3.2% and 21.2%,



(a) STP (b) Normalized ANTT

Figure 44. Effectiveness of QBMI and DMIL with different L1 D-cache capacities, on top of Warped-Slicer.



(a) STP (b) Normalized ANTT

Figure 43. Effectiveness of QBMI and DMIL with LRR warp scheduling policy, on top of Warped-Slicer.

respectively; in terms of ANTT, WS-QBMI and WS-DMIL outperform WS by 3.2% and 17.2%.

4.4.4 Hardware Overhead

As described in Section 4.3.3, the hardware cost for a memory instruction limiting number generator (MILG) includes one 7-bit inflight memory instruction counter (maximum 128 instructions can access L1 D-cache concurrently), one 12-bit reservation failure counter, one 10-bit memory request counter, and one 10-bit right shifter (only wires). For QBMI, one more 10-bit memory instruction counter and extra arithmetic logics are required to compute 'Req/Minst' and quotas. Although the amount of these components are proportional to the

number of SMs, those overheads are negligible, compared to the area of a GPU [67][68][69][70].

4.4.5 Further Discussion

In this part, we further discuss the inadequacy to partitioning cache-miss-related resources as well as the energy efficiency and applicability of our proposed schemes.

Although it is tempting to partition cache-miss-related resources to prevent any kernel from starvation in allocating them, our experiments show that simply partitioning such resources cannot improve performance. This is because all accesses to LSU are in-order and even when a kernel's assigned portion of resources is not fully occupied, its accesses can be blocked by accesses from other co-run kernels of which the assigned resources already saturate, leading to unrelieved memory pipeline stalls.

Regarding energy efficiency, with our proposed schemes, although the average dynamic power may increase due to the improved computing resource utilization, the overall energy efficiency is improved due to much reduced leakage energy.

Although just the metric *reservation failures per memory request* is used to generate memory instruction limiting numbers, the underlying idea can be applied to other parts of the memory access path. For example, stalls encountered at the L1-interconnect and/or interconnect-L2 queues, can be incorporated to obtain memory instruction limiting numbers. And we left the comprehensive investigation in future work.

4.5 Related Work

Resource distribution among threads on CPUs: Several studies have addressed resource distribution among threads in simultaneous multithreading on CPUs, with a focus on cache partitioning [19][49][64][36]. Our work targets concurrent kernel execution on GPUs where there are large numbers of memory accesses due to massive multithreading and cache partitioning is not as effective as for CPUs. Therefore, we propose balanced memory request issuing and inflight memory instructions limiting to prevent one kernel starving from failing to access LSU and reduce LSU stalls.

Concurrent kernel execution on GPUs: Several software-centric GPU multiprogramming approaches have been studied as discussed in Section 4.1.

Researchers have also proposed hardware schemes to better exploit concurrent kernel execution on GPUs. Adriaens et al. [2] proposed spatial multitasking to groups SMs into different sets which can run different kernels. Ukidave et al. [61] studied runtime support for adaptive spatial partition on GPUs. Aguilera et al. [3] showed the unfairness of the spatial multitasking and proposed fair resource allocation for both performance and fairness. Tanasic el al. [58] proposed pre-emption mechanisms to allow dynamic spatial sharing of GPU cores across kernels. Gregg et al. [16] proposed a kernel scheduler to increase throughput. Wang et al. [63] proposed dynamic thread block launching to better support irregular applications. Wang et al. [65] and Xu et al. [71] addressed SM resource partition at the granularity of thread block. In comparison, we focus on mitigating interference among kernels and further improving resource utilization within an SM.

Thread throttling and cache management on GPUs: Managing accesses to the limited memory resources has been a challenge on GPUs. Guz et al. [17] showed that a performance valley exists with increased number of threads accessing a cache. Bakhoda et al. [5] showed some applications perform better when scheduling fewer TBs. Kayıran et al. [28] and Xie et al. [72] dynamically adjust the number of TBs accessing L1 D-caches. Rogers et al. [52] proposed CCWS to control the number of warps scheduled.

On GPU cache management, Jia et al. [24] used multiple queues to preserve intra-warp locality. Kloosterman et al. [30] proposed WarpPool to exploit inter-warp locality with request queues. Detecting and protecting hot cache lines has been proposed in [34].

Researchers have also exploited the combination of thread throttling and cache bypassing. Li et al. [36] proposed priority-based cache allocation on top of CCWS. Chen et al. proposed CBWT [7] to adopt PDP for L1 D-cache bypassing and applies warp throttling. Li et al. [33] propose a compile-time framework for cache bypassing at the warp level.

These approaches mainly target cache locality. However, as Sethia et al. [54] and Dai et al. [10] demonstrated, cache-miss-related resource saturation can cause severe memory pipeline

stalls and performance degradation. To address this issue, they proposed Mascar and MDB, respectively.

In comparison, our schemes do not throttle any TBs or warps but limit the number of in-flight memory instructions to reduce memory pipeline stalls and accelerate concurrent kernel execution with one SM shared by multiple kernels. Our approaches are complementary to cache bypassing: if not controlled, bypassing can make a memory-intensive kernel occupy even more memory resources.

4.6 Conclusions

In this paper, we show that the state-of-art intra-SM sharing techniques do not address the interference in CKE on GPUs. We argue that dedicated management on memory access is necessary and propose to balance memory request issuing from individual kernels and limit inflight memory instructions to mitigate memory pipeline stalls. We evaluated our proposed schemes on two intra-SM sharing schemes, Warped-Slicer and SMK. The experimental results show that our proposed approaches improve system throughput by 26.5% and 27.2% on average over Warped-Slicer and SMK, respectively, with minor hardware cost. Our approaches also significantly improve the fairness.

Chapter 5

Conclusion

In this dissertation, we identify a few inefficiencies existent in current GPU memory architecture and propose optimizations to improve performance and energy efficiency for both single kernel execution and concurrent kernel execution. We also advocate a sound baseline for GPU memory architecture research.

First, we demonstrate that although throughput-oriented GPUs hide long operation latency with massive multithreading, limited per thread cache capacity and massive memory requests can easily cause cache thrashing and memory pipeline stalls. Therefore, we propose a performance model for L1 D-cache contention and cache-miss-related resource congestion. Based on the model, we design a cost-effective dynamic warp/TB level GPU cache bypassing scheme. The experimental results show that our scheme achieves significant performance improvement over the baseline and outperforms the state-of-the-art GPU cache management schemes. We also demonstrate that our scheme remains effective with different cache capacities, various warp scheduling policies and altered application input sizes.

Second, we comprehensively investigate the performance impact of difference design choices in various aspects of GPU memory hierarchy. Our studies show that advanced cache indexing functions should be deployed in the first place to reduce the severe conflict misses; allocate-on-fill should be used to increase cache hits and reduce memory pipeline stalls; the number of MSHRs plays an important role in affecting the cache efficiency besides supporting MLP/TLP. Furthermore, we show that a good memory partition mapping function, such as Xor, should be deployed to mitigate the problem of memory camping. And while previous GPU cache bypassing works unrealistically assume an unlimited number of in-flight bypassed requests can be supported, we demonstrate such a constraint can significantly affect the performance of a GPU cache bypassing scheme and this factor should be taken into account in GPU cache bypassing studies. Finally, we propose the sound baseline configuration for future GPU memory architecture studies and open source it.

Third, we show that although intra-SM sharing has been proposed to improve intra-SM resource utilization, the overall performance may be undermined in the state-of-the-art intra-SM sharing schemes, due to the interference among concurrent kernels. Therefore, we argue that dedicated management on memory access is necessary and propose to balance memory request issuing from individual kernels and limit inflight memory instructions to mitigate memory pipeline stalls. The experimental results show that our proposed approaches can effectively improve the system throughput and fairness of two state-of-the-art intra-SM sharing schemes with lightweight hardware cost.

REFERENCES

[1] AMD GCN Architecture White paper, 2012.

[2] Adriaens, Jacob T., Katherine Compton, Nam Sung Kim, and Michael J. Schulte. "The case for GPGPU spatial multitasking." In IEEE International Symposium on High-Performance Comp Architecture, pp. 1-12. IEEE, 2012.

[3] Aguilera, Paula, Katherine Morrow, and Nam Sung Kim. "Fair share: Allocation of GPU resources for both performance and fairness." In 2014 IEEE 32nd International Conference on Computer Design (ICCD), pp. 440-447. IEEE, 2014.

[4] Awatramani, Mihir, Joseph Zambreno, and Diane Rover. "Increasing GPU throughput using kernel interleaved thread block scheduling." In 2013 IEEE 31st International Conference on Computer Design (ICCD), pp. 503-506. IEEE, 2013.

[5] Bakhoda, Ali, George L. Yuan, Wilson WL Fung, Henry Wong, and Tor M. Aamodt. "Analyzing CUDA workloads using a detailed GPU simulator." InPerformance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on, pp. 163-174. IEEE, 2009.

[6] Burtscher, Martin, Rupesh Nasre, and Keshav Pingali. "A quantitative study of irregular programs on GPUs." In Workload Characterization (IISWC), 2012 IEEE International Symposium on, pp. 141-151. IEEE, 2012.

[7] Chen, Xuhao, Li-Wen Chang, Christopher I. Rodrigues, Jie Lv, Zhiying Wang, and Wen-Mei Hwu. "Adaptive cache management for energy-efficient gpu computing." In Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 343-355. IEEE Computer Society, 2014.

[8] Che, Shuai, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. "Rodinia: A benchmark suite for heterogeneous computing." In Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on, pp. 44-54. IEEE, 2009.

[9] Chow, C. K. "Determination of cache's capacity and its matching storage hierarchy." IEEE Transactions on Computers 100, no. 2 (1976): 157-164.

[10] Dai, Hongwen, S. Gupta, C. Li, C. Kartsaklis, M. Mantor, and H. Zhou. "A model-driven approach to warp/thread-block level GPU cache bypassing." InProceedings of the Design Automation Conference (DAC), Austin, TX, USA, pp. 5-9. 2016.

[11] Duong, Nam, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V. Veidenbaum. "Improving cache management policies using dynamic reuse distances." In Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on, pp. 389-400. IEEE, 2012.

[12] Eyerman, Stijn, and Lieven Eeckhout. "Restating the case for weighted-ipc metrics to evaluate multiprogram workload performance." IEEE Computer Architecture Letters 13, no. 2 (2014): 93-96.

[13] Gaur, Jayesh, Mainak Chaudhuri, and Sreenivas Subramoney. "Bypass and insertion algorithms for exclusive last-level caches." In Computer Architecture (ISCA), 2011 38th Annual International Symposium on, pp. 81-92. IEEE, 2011.

[14] González, Antonio, Mateo Valero, Nigel Topham, and Joan M. Parcerisa. "Eliminating cache conflict misses through XOR-based placement functions." In Proceedings of the 11th international conference on Supercomputing, pp. 76-83. ACM, 1997.

[15] Grauer-Gray, Scott, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. "Auto-tuning a high-level language targeted to GPU codes." In Innovative Parallel Computing (InPar), 2012, pp. 1-10. IEEE, 2012.

[16] Gregg, Chris, Jonathan Dorn, Kim Hazelwood, and Kevin Skadron. "Fine-grained resource sharing for concurrent GPGPU kernels." In Presented as part of the 4th USENIX Workshop on Hot Topics in Parallelism. 2012.

[17] Guz, Zvika, Evgeny Bolotin, Idit Keidar, Avinoam Kolodny, Avi Mendelson, and Uri C. Weiser. "Many-core vs. many-thread machines: Stay away from the valley." IEEE Computer Architecture Letters 8, no. 1 (2009): 25-28.

[18] Hartstein, Allan, Viji Srinivasan, Thomas R. Puzak, and Philip G. Emma. "Cache miss behavior: is it√ 2?." In Proceedings of the 3rd conference on Computing frontiers, pp. 313-320. ACM, 2006.

[19] Herdrich, Andrew, Edwin Verplanke, Priya Autee, Ramesh Illikkal, Chris Gianos, Ronak Singhal, and Ravi Iyer. "Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2600 v3 product family." In 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 657-668. IEEE, 2016.

[20] Hong, Sunpyo, and Hyesoon Kim. "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness." In ACM SIGARCH Computer Architecture News, vol. 37, no. 3, pp. 152-163. ACM, 2009.

[21] Huang, Jen-Cheng, Joo Hwan Lee, Hyesoon Kim, and Hsien-Hsin S. Lee. "Gpumech: Gpu performance modeling technique based on interval analysis." In Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 268-279. IEEE Computer Society, 2014.

[22] Jiao, Qing, Mian Lu, Huynh Phung Huynh, and Tulika Mitra. "Improving GPGPU energy-efficiency through concurrent kernel execution and DVFS." In Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, pp. 1-11. IEEE Computer Society, 2015.

[23] Jia, Wenhao, Kelly A. Shaw, and Margaret Martonosi. "MRPB: Memory request prioritization for massively parallel processors." In 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), pp. 272-283. IEEE, 2014.

[24] Jog, Adwait, Evgeny Bolotin, Zvika Guz, Mike Parker, Stephen W. Keckler, Mahmut T. Kandemir, and Chita R. Das. "Application-aware memory system for fair and efficient execution of concurrent GPGPU applications." InProceedings of workshop on general purpose processing using GPUs, p. 1. ACM, 2014.

[25] Jog, Adwait, Onur Kayiran, Tuba Kesten, Ashutosh Pattnaik, Evgeny Bolotin, Niladrish Chatterjee, Stephen W. Keckler, Mahmut T. Kandemir, and Chita R. Das. "Anatomy of GPU Memory System for Multi-Application Execution." In Proceedings of the 2015 International Symposium on Memory Systems, pp. 223-234. ACM, 2015.

[26] Jones, Stephen. "Introduction to dynamic parallelism." In GPU Technology Conference Presentation S, vol. 338, p. 2012. 2012.

[27] Justin Luitjens. “Cuda Streams: Best Practices and Common Pitfalls”, GPU Techonology Conference, 2015.

[28] Kayıran, Onur, Adwait Jog, Mahmut Taylan Kandemir, and Chita Ranjan Das. "Neither more nor less: optimizing thread-level parallelism for GPGPUs." InProceedings of the 22nd international conference on Parallel architectures and compilation techniques, pp. 157-166. IEEE Press, 2013.

[29] Kharbutli, Mazen, Keith Irwin, Yan Solihin, and Jaejin Lee. "Using prime numbers for cache indexing to eliminate conflict misses." In Software, IEE Proceedings-, pp. 288-299. IEEE, 2004.

[30] Kloosterman, John, Jonathan Beaumont, Mick Wollman, Ankit Sethia, Ron Dreslinski, Trevor Mudge, and Scott Mahlke. "WarpPool: sharing requests with inter-warp coalescing for throughput processors." In Proceedings of the 48th International Symposium on Microarchitecture, pp. 433-444. ACM, 2015.

[31] Kroft, David. "Lockup-free instruction fetch/prefetch cache organization." In Proceedings of the 8th annual symposium on Computer Architecture, pp. 81-87. IEEE Computer Society Press, 1981.

[32] Leng, Jingwen, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. "GPUWattch: enabling energy optimizations in GPGPUs." In ACM SIGARCH Computer Architecture News, vol. 41, no. 3, pp. 487-498. ACM, 2013.

[33] Li, Ang, Gert-Jan van den Braak, Akash Kumar, and Henk Corporaal. "Adaptive and transparent cache bypassing for GPUs." In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, p. 17. ACM, 2015.

[34] Li, Chao, Shuaiwen Leon Song, Hongwen Dai, Albert Sidelnik, Siva Kumar Sastry Hari, and Huiyang Zhou. "Locality-driven dynamic GPU cache bypassing." In Proceedings of the 29th ACM on International Conference on Supercomputing, pp. 67-77. ACM, 2015.

[35] Li, Chao, Yi Yang, Hongwen Dai, Shengen Yan, Frank Mueller, and Huiyang Zhou. "Understanding the tradeoffs between software-managed vs. hardware-managed caches in

GPUs." In Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on, pp. 231-242. IEEE, 2014.

[36] Li, Dong, Minsoo Rhu, Daniel R. Johnson, Mike O'Connor, Mattan Erez, Doug Burger, Donald S. Fussell, and Stephen W. Redder. "Priority-based cache allocation in throughput processors." In 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), pp. 89-100. IEEE, 2015.

[37] Liang, Yun, Huynh Phung Huynh, Kyle Rupnow, Rick Siow Mong Goh, and Deming Chen. "Efficient gpu spatial-temporal multitasking." IEEE Transactions on Parallel and Distributed Systems 26, no. 3 (2015): 748-760.

[38] Ma, Yi, Hongliang Gao, and Huiyang Zhou. "Using indexing functions to reduce conflict aliasing in branch prediction tables." IEEE Transactions on Computers 55, no. 8 (2006): 1057-1061.

[39] Narasiman, Veynu, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. "Improving GPU performance via large warps and two-level warp scheduling." In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 308-317. ACM, 2011.

[40] Nugteren, Cedric, Gert-Jan van den Braak, Henk Corporaal, and Henri Bal. "A detailed GPU cache model based on reuse distance theory." In High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on, pp. 37-48. IEEE, 2014.

[41] NVIDIA, “CUDA C/C++ SDK code samples,”2011.

[42] NVIDIA’s CUDA compute architecture: Fermi. 2009.

[43] NVIDIA Kepler GK110 white paper. 2012.

[44] NVIDIA Maxwell GM204 Architecture Whitepaper, 2014.

[45] NVIDIA Parallel Thread Execution ISA Version 4.2.

[46] NVIDIA Pascal GP100 Architecture, GTC, 2016.

[47] O'Neil, Molly A., and Martin Burtscher. "Microarchitectural performance characterization of irregular GPU kernels." In Workload Characterization (IISWC), 2014 IEEE International Symposium on, pp. 130-139. IEEE, 2014.

[48] Pai, Sreepathi, Matthew J. Thazhuthaveetil, and Ramaswamy Govindarajan. "Improving GPGPU concurrency with elastic kernels." ACM SIGPLAN Notices 48, no. 4 (2013): 407-418.

[49] Qureshi, Moinuddin K., and Yale N. Patt. "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches." In Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 423-432. IEEE Computer Society, 2006.

[50] Rau, B. Ramakrishna. "Pseudo-randomly interleaved memory." In ACM SIGARCH Computer Architecture News, vol. 19, no. 3, pp. 74-83. ACM, 1991.

[51] Rogers, Phil, Ben Sander, Yeh-Ching Chung, B. R. Gaster, H. Persson, and W-M. W. Hwu. "Heterogeneous system architecture (hsa): Architecture and algorithms tutorial." In Proceedings of the 41st Annual International Symposium on Computer Architecture.

[52] Rogers, Timothy G., Mike O'Connor, and Tor M. Aamodt. "Cache-conscious wavefront scheduling." In Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 72-83. IEEE Computer Society, 2012.

[53] Schulte, Michael J., Mike IgnatoWSki, Gabriel H. Loh, Bradford M. Beckmann, William C. Brantley, Sudhanva Gurumurthi, Nuwan Jayasena, Indrani Paul, Steven K. Reinhardt, and Gregory Rodgers. "Achieving exascale capabilities through heterogeneous computing." IEEE Micro 35, no. 4 (2015): 26-36.

[54] Sethia, Ankit, D. Anoushe Jamshidi, and Scott Mahlke. "Mascar: Speeding up gpu warps by reducing memory pitstops." In 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), pp. 174-185. IEEE, 2015.

[55] Sethia, Ankit, and Scott Mahlke. "Equalizer: Dynamic tuning of gpu resources for efficient execution." In Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 647-658. IEEE Computer Society, 2014.

[56] Stratton, John A., Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W. Hwu. "Parboil: A revised benchmark suite for scientific and commercial throughput computing." Center for Reliable and High-Performance Computing 127 (2012).

[57] Singh, Inderpreet, Arrvindh Shriraman, Wilson WL Fung, Mike O'Connor, and Tor M. Aamodt. "Cache coherence for GPU architectures." In High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on, pp. 578-590. IEEE, 2013.

[58] Tanasic, Ivan, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. "Enabling preemptive multiprogramming on GPUs." In ACM SIGARCH Computer Architecture NeWS, vol. 42, no. 3, pp. 193-204. IEEE Press, 2014.

[59] Tian, Yingying, Sooraj Puthoor, Joseph L. Greathouse, Bradford M. Beckmann, and Daniel A. Jiménez. "Adaptive GPU cache bypassing." In Proceedings of the 8th Workshop on General Purpose Processing using GPUs, pp. 25-35. ACM, 2015.

[60] Tuck, James, Luis Ceze, and Josep Torrellas. "Scalable cache miss handling for high memory-level parallelism." In Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 409-422. IEEE Computer Society, 2006.

[61] Ukidave, Yash, Charu Kalra, David Kaeli, Perhaad Mistry, and Dana Schaa. "Runtime support for adaptive spatial partitioning and inter-kernel communication on gpus." In Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on, pp. 168-175. IEEE, 2014.

[62] Wang, Bin, Zhuo Liu, Xinning Wang, and Weikuan Yu. "Eliminating intra-warp conflict misses in GPU." In Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, pp. 689-694. EDA Consortium, 2015.

[63] Wang, Jin, Norm Rubin, Albert Sidelnik, and Sudhakar Yalamanchili. "Dynamic thread block launch: a lightweight execution mechanism to support irregular applications on GPUs." In ACM SIGARCH Computer Architecture NeWS, vol. 43, no. 3, pp. 528-540. ACM, 2015.

[64] Wang, Ruisheng, and Lizhong Chen. "Futility scaling: High-associativity cache partitioning." In 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 356-367. IEEE, 2014.

[65] Wang, Zhenning, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. "Simultaneous Multikernel GPU: Multi-tasking throughput processors via fine-grained

sharing." In 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 358-369. IEEE, 2016.

[66] Wilton, Steven JE, and Norman P. Jouppi. "CACTI: An enhanced cache access and cycle time model." IEEE Journal of Solid-State Circuits 31, no. 5 (1996): 677-688.

[67] Whitepaper: NVIDIA's Next Generation CUDA TM Compute Architecture: Kepler TM GK110, tech. rep., NVIDIA, 2012.

[68] Whitepaper: NVIDIA's Next Generation CUDA TM Compute Architecture: Fermi TM , tech. rep., NVIDIA, 2009.

[69] Whitepaper: NVIDIA GeForce GTX980, tech. rep., NVIDIA, 2014.

[70] Whitepaper: NVIDIA GeForce GTX1080, tech. rep., NVIDIA, 2016.

[71] Xu, Qiumin, Hyeran Jeon, Keunsoo Kim, Won Woo Ro, and Murali Annavaram. "Warped-slicer: efficient intra-SM slicing through dynamic resource partitioning for GPU multiprogramming." In Proceedings of the 43rd International Symposium on Computer Architecture, pp. 230-242. IEEE Press, 2016.

[72] Xie, Xiaolong, Yun Liang, Yu Wang, Guangyu Sun, and Tao Wang. "Coordinated static and dynamic cache bypassing for GPUs." In 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), pp. 76-88. IEEE, 2015.

[73] Xie, Yuejian, and Gabriel H. Loh. "PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches." In ACM SIGARCH Computer Architecture News, vol. 37, no. 3, pp. 174-183. ACM, 2009.

[74] Zhang, Yao, and John D. Owens. "A quantitative performance analysis model for GPU architectures." In High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on, pp. 382-393. IEEE, 2011.

[75] Zhong, Jianlong, and Bingsheng He. "Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling." IEEE Transactions on Parallel and Distributed Systems 25, no. 6 (2014): 1522-1532.